Brigitte Pientka
Cesare Tinelli (Eds.)

LNAI 14132

Automated Deduction – CADE 29

29th International Conference on Automated Deduction
Rome, Italy, July 1–4, 2023
Proceedings

Springer

Lecture Notes in Computer Science

Lecture Notes in Artificial Intelligence 14132

Founding Editor

Jörg Siekmann

The series Lecture Notes in Artificial Intelligence (LNAI) was established in 1988 as a topical subseries of LNCS devoted to artificial intelligence.

The series publishes state-of-the-art research results at a high level. As with the LNCS mother series, the mission of the series is to serve the international R & D community by providing an invaluable service, mainly focused on the publication of conference and workshop proceedings and postproceedings.

Brigitte Pientka · Cesare Tinelli
Editors

Automated Deduction – CADE 29

29th International Conference on Automated Deduction
Rome, Italy, July 1–4, 2023
Proceedings

Springer

Editors

Brigitte Pientka
McGill University
Montreal, QC, Canada

Cesare Tinelli
The University of Iowa
Iowa City, IA, USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Artificial Intelligence
ISBN 978-3-031-38498-1 ISBN 978-3-031-38499-8 (eBook)
https://doi.org/10.1007/978-3-031-38499-8

LNCS Sublibrary: SL7 – Artificial Intelligence

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

This volume contains the proceedings of the 29th International Conference on Automated Deduction (CADE-29). CADE is the major forum for the presentation of research in all aspects of automated deduction, including foundations, applications, implementations, and practical experience. CADE-29 was held on 1–4 July 2023, hosted at the Faculty of Civil and Industrial Engineering of the Sapienza University of Rome, Italy, and co-located with the 8th International Conference on Formal Structures for Computation and Deduction (FSCD). CADE-29 emphasized the breadth of topics that are of interest, including applications in and beyond computer science and mathematics, and the use/contribution of automated deduction in AI.

The Program Committee (PC) examined 74 submissions this year and decided to accept 33 of them (28 full papers and 5 short papers or system descriptions). Submissions were single-blind and each of them was reviewed by at least three PC members or their external reviewers. The criteria for evaluation were originality and significance, technical quality, comparison with related work, quality of presentation, and reproducibility of experiments.

The program of the conference included three invited talks, two of which were joint talks with FSCD:

– "Lambda-Superposition: From Theory to Trophy" by *Jasmin Blanchette*, Ludwig-Maximilians-Universität München, Germany
– "Nominal Techniques for Software Specification and Verification" by *Maribel Fernandez*, King's College London, UK (joint talk)
– "Can we trust AI?" by *Mateja Jamnik*, University of Cambridge, UK (joint talk)

A fourth invited talk, "Automated Reasoning with Data," was given by *Moshe Vardi* as recipient of the 2023 Herbrand Award.

The conference hosted several workshops, and one competition on July 4–6:

– **ADeMaL**: Automated Deduction for Machine Learning
– **Vampire 2023**: The 7th Vampire Workshop
– **ThEdu'23**: Theorem Proving Components for Educational Software
– **SMT 2023**: The 21st International Workshop on Satisfiability Modulo Theories
– **CASC 2023**: The CADE ATP System Competition

In addition to the best paper awards, three CADE awards were presented at the conference:

– The 2023 Herbrand Award for Distinguished Contributions to Automated Reasoning, awarded to Moshe Y. Vardi, Rice University, USA, in recognition of his many foundational contributions to logic and automated reasoning, in particular automata-based verification methods, constraint solving, and knowledge representation.
– The Thoralf Skolem Award for CADE papers that have passed the test of time by being the most influential papers in the field, awarded to each of the following papers:

  - “Deciding Combinations of Theories” by *Robert E. Shostak*, CADE-6 (1982)
  - “The TPTP Problem Library” by *Geoff Sutcliffe, Christian B. Suttner, and Theodor Yemenis*, CADE-12 (1994)
  - “A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions” by *Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani*, CADE-18 (2002)
  - “The Tree Width of Separation Logic with Recursive Definitions” by *Radu Iosif, Adam Rogalewicz, and Jirí Simácek*, CADE-24 (2013)

- The Bill McCune PhD Award for a PhD thesis’ substantive contributions to the field of Automated Reasoning, awarded to Alessandro Gianola, Free University of Bozen-Bolzano, Italy.

Sincere thanks go to the many people who contributed to the success of CADE-29 — the authors, the participants, the invited speakers, the members of the PC, the external subreviewers, the general chair, the workshop and tutorial chair, the publicity chair, the staff at Springer, and the EasyChair team.

CADE-29 gratefully acknowledges the support of the CADE trustees, the board of the Association for Automated Reasoning, ACM SIGLOG, and the sponsors Amazon Web Services and Springer.

July 2023

Brigitte Pientka
Cesare Tinelli

Organization

General Chair

Daniele Gorla	Sapienza University of Rome, Italy

Workshop and Tutorial Chair

Ivano Salvo	Sapienza University of Rome, Italy

Publicity Chair

Haniel Barbosa	Universidade Federal de Minas Gerais, Brazil

Program Committee Chairs

Brigitte Pientka	McGill University, Canada
Cesare Tinelli	University of Iowa, USA

Program Committee

Carlos Areces	Universidad Nacional de Córdoba, Argentina
Jeremy Avigad	Carnegie Mellon University, USA
David Baelde	ENS Paris-Saclay & Inria Paris, France
Maria Paola Bonacina	Università degli Studi di Verona, Italy
Cyril Cohen	Inria Sophia Antipolis - Méditerranée, France
Liron Cohen	Ben-Gurion University, Israel
Clare Dixon	University of Manchester, UK
Carsten Fuhs	Birkbeck, University of London, UK
Alberto Griggio	Fondazione Bruno Kessler, Italy
Stefan Hetzl	Vienna University of Technology, Austria
Marijn Heule	Carnegie Mellon University, USA
Nao Hirokawa	Japan Advanced Institute of Science and Technology, Japan
Mikolas Janota	Czech Technical University in Prague, Czechia
Matti Järvisalo	University of Helsinki, Finland
Cezary Kaliszyk	University of Innsbruck, Austria

Konstantin Korovin	University of Manchester, UK
Peter Lammich	TU Munich, Germany
Aina Niemetz	Stanford University, USA
Naoki Nishida	Nagoya University, Japan
Nicola Olivetti	Aix-Marseille University, France
Lawrence Paulson	University of Cambridge, UK
Brigitte Pientka	McGill University, Canada
Andrei Popescu	University of Sheffield, UK
Florian Rabe	FAU Erlangen-Nürnberg, Germany
Giles Reger	AWS and University of Manchester, UK
Cody Roux	Amazon Web Services, USA
Uli Sattler	University of Manchester, UK
Stephan Schulz	DHBW Stuttgart, Germany
Martina Seidl	Johannes Kepler University Linz, Austria
Viorica Sofronie-Stokkermans	University of Koblenz, Germany
Alexander Steen	University of Greifswald, Germany
Martin Suda	Czech Technical University in Prague, Czechia
Enrico Tassi	Inria, France
Cesare Tinelli	University of Iowa, USA
Sophie Tourret	Inria, France and MPI for Informatics, Germany
Uwe Waldmann	MPI for Informatics, Germany
Sarah Winkler	Free University of Bozen-Bolzano, Italy
Yoni Zohar	Bar-Ilan University, Israel

Additional Reviewers

Erika Ábrahám
Antonis Achilleos
Takahito Aoto
François Bobot
James Brotherston
Chad Brown
Guillaume Burel
Filip Bártek
David Cerna
Kaustuv Chaudhuri
Md Solimul Chowdhury
Gabriel Ebner
Raul Fervari
Pascal Fontaine
Thibault Gauthier
Khalil Ghorbal
Alessandro Gianola
Stéphane Graham-Lengrand
Jonathan Huerta y Munive
Sohei Ito
Jan Jakubuv
Albert Jiang
Ariel Kellison
Patrick Koopmann
Temur Kutsia
Dennis Müller
Jakob Nordström
Miroslav Olšák
Eugenio Orlandelli
Pedro Orvalho
Nicolas Peltier
Bartosz Piotrowski

Gian Luca Pozzato
Mathias Preiner
Stanisław Purgał
Gianluca Redondi
Joseph Reeves
Martin Riener
Colin Rothgang
Navid Roux
Reuben Rowe
Claudio Sacerdoti Coen
Luca San Mauro
Jan Frederik Schaefer
Tanja Schindler
Anders Schlichtkrull
Ying Sheng
Nicholas Smallbone
Guilherme Toledo
Dmitriy Traytel
Makarius Wenzel
Yechuan Xia
Akihisa Yamada
Emre Yolcu

Board of CADE Trustees

Board of the Association for Automated Reasoning

Sponsors

aws

Springer

Invited Talks

λ-Superposition: From Theory to Trophy

Jasmin Blanchette[1,2,3]

[1]Ludwig-Maximilians-Universität München, Munich, Germany
jasmin.blanchette@lmu.de
[2]Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrüücken, Germany
[3]Université de Lorraine, CNRS, Inria, LORIA, Nancy, France

This extended abstract describes work performed in collaboration with Alexander Bentkamp, Simon Cruanes, Visa Nummelin, Stephan Schulz, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann on the design and implementation of λ-superposition, in the context of the Matroyshka research project.

When I conceived Matroyshka in 2015, my ambition was to develop higherorder provers that perform well on higher-order proof obligations originating from Isabelle/HOL [11] and other proof assistants. Lawrence Paulson had noticed that the performance on truly higher-order goals left much to be desired and "given the inherent difficulty of performing higher-order reasoning using first-order theorem provers, the way forward is to integrate Sledgehammer with an actual higher-order theorem prover, such as LEO-II" [13]. However, the subsequent integration of LEO-II [4] and Satallax [7] failed to bring the expected benefits [16]. My hypothesis was that most Isabelle problems have a large first-order component and the existing higher-order provers were not optimized for this kind of reasoning.

To obtain higher-order provers that excel at first-order reasoning, I proposed to start with a highly successful first-order calculus, superposition, and generalize it, as much as possible, in a graceful way, culminating with a higher-order calculus. Provers implementing this calculus would combine the strengths of native higher-order provers and the strengths of the superposition provers that served as Sledgehammer backends: E [14], SPASS [6], and Vampire [5].

To tackle the challenge of designing this calculus, which we call λ-superposition, we identified three milestones that we reached in turn. We first designed a superposition-like calculus for a λ-free, Boolean-free higher-order logic (also called applicative first-order logic) [1]. This logic supports partial application of function symbols (e.g., f or $\mathsf{f}\ \mathsf{a}$, where f is binary) and application of variables (e.g., $y\ \mathsf{a}$). Already at this stage, the first serious issue arose with the term order that superposition uses to prune the search space. We were able to work around the issue by introducing a new inference rule called argument congruence. For this and the other milestones, much of the work went into ensuring refutational completeness.

For the second milestone, we designed a superposition-like calculus for a logic that supports λ-abstractions but not interpreted Booleans [3]. One difficulty that arose is that inferences need to perform higher-order unification. Unfortunately, higher-order unification is ill-behaved: It is undecidable and can yield a possibly infinite stream of unifiers. Moreover, due to interactions with the term order, we need to perform full unification (including flex-flex pairs) [17] and not simply preunification [10].

For the third milestone, we added support for interpreted Booleans [2]. This step was based on ideas by Ganzinger and Stuber [9]. They showed how to support logical symbols inside a superposition-like calculus, but fell short of including an interpreted Boolean type. Thus, we extended Ganzinger and Stuber's work [12] and used it as the basis of a graceful generalization to higher-order logic.

Whenever we designed a calculus, we also made sure to implement it in the Zipperposition prover [8]. Zipperposition was originally developed by Cruanes to explore induction, arithmetic, and deduction modulo. It is written in OCaml and is highly extensible. He extended it with a pragmatic higher-order mode with support for λ-abstractions and extensionality, without any completeness guarantees. This mode formed the basis for our subsequent work. Empirical evaluations on TPTP and Sledgehammer benchmarks were initially disappointing, but after some extensive tuning and new ideas for heuristics, Zipperposition became highly competitive, finishing first in the higher-order theorem division of the CADE ATP System Competition (CASC) in 2020, 2021, and 2022. Inspired by a similar integration in Leo-III [15] and Satallax, Zipperposition incorporates E as a backend to tackle first-order subproblems.

We also implemented λ-superposition in the high-performance prover E [18,19]. The E implementation is pragmatic and sacrifices completeness. For example, the possibly infinite stream of unifiers is truncated to make it finite, and some of the most explosive rules of λ-superposition are omitted. Probably because Zipperposition has a portfolio of modes extensively tuned against the TPTP library and uses a version of E as a backend, E finished only second in the higher-order theorem division of CASC 2022. On the other hand, E finished first in the Sledgehammer division of the same competition. Despite this, the performance improvement over Sledgehammer's first-order backends is small. I suspect that Isabelle problems are even more first-order than I thought.

We learned a few other lessons in the process:

- The identification of reasonable milestones was invaluable.
- The completeness proofs gave us some useful guidance as we designed the calculi, even if it turns out that the best empirical modes are incomplete.
- Another useful guide was the design goal of achieving, as much as possible, a graceful generalization, preserving the features that make standard superposition successful on first-order problems.
- Disappointing evaluations can simply mean that more fine-tuning and heuristics are needed.
- The presence of many complementary modes in a well-tuned portfolio can be as important as a highly efficient implementation.

Acknowledgment. I thank Alexander Bentkamp, Stephan Schulz, Mark Summerfield, Petar Vukmirović, and Uwe Waldmann for textual suggestions. This research has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant No. 713999, Matryoshka). This research has also received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

References

1. Bentkamp, A., Blanchette, J., Cruanes, S., Waldmann, U.: Superposition for lambda-free higher-order logic. Log. Meth. Comput. Sci. **17**(2), 1:11:38 (2021)
2. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P.: Superposition for higher-order logic. J. Autom. Reason. **67**(1), article 10 (2023)
3. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P., Waldmann, U.: Superposition with lambdas. J. Autom. Reason. 65(7), 893–940 (2021)
4. Benzmüller, C., Paulson, L.C., Theiss, F., Fietzke, A.: LEO-II—A cooperative automatic theorem prover for higher-order logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS, vol. 5195, pp. 162–170. Springer Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_14
5. Bhayat, A., Reger, G.: A combinator-based superposition calculus for higher-order logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020, Part I. LNCS, vol. 12166, pp. 278–296. Springer Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_16
6. Blanchette, J.C., Popescu, A., Wand, D., Weidenbach, C.: More SPASS with Isabelle: Superposition with hard sorts and configurable simplification. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 345–360. Springer Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_24
7. Brown, C.E.: Satallax: An automatic higher-order prover. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 111–117. Springer Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_11
8. Cruanes, S.: Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond. PhD thesis, École polytechnique (2015)
9. Ganzinger, H., Stuber, J.: Superposition with equivalence reasoning and delayed clause normal form transformation. Inform. Comput. **199**(1–2), 3–23 (2005)
10. Huet, G.P.: A unification algorithm for typed lambda-calculus. Theor. Comput. Sci. **1**(1), 27–57 (1975)
11. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
12. Nummelin, V., Bentkamp, A., Tourret, S., Vukmirović, P.: Superposition with first-class Booleans and inprocessing clausification. In: Platzer, A., Sutcliffe, G. (eds.) CADE-28. LNCS, Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_22
13. Paulson, L.C.: Three years of experience with Sledgehammer, a practical link between automated and interactive theorem provers. In: Konev, B., Schmidt, R., Schulz, S. (eds.) PAAR-2010 (2010)
14. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) CADE-27. LNCS, vol. 11716, pp. 495–507. Springer Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_29
15. Steen, A., Benzmüller, C.: The higher-order prover Leo-III. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS, vol. 10900, pp. 108–116. Springer Cham (2018).https://doi.org/10.1007/978-3-319-94205-6_8

16. Sultana, N., Blanchette, J.C., Paulson, L.C.: LEO-II and Satallax on the Sledgehammer test bench. J. Appl. Log. **11**(1), 91–102 (2013)
17. Vukmirović, P., Bentkamp, A., Nummelin, V.: Efficient full higher-order unification.In: Ariola, Z.M. (ed.) FSCD 2020. LIPIcs, vol. 167, pp. 5:1–5:17. Schloss Dagstuhl—Leibniz-Zentrum für Informatik (2020)
18. Vukmirović, P., Blanchette, J., Cruanes, S., Schulz, S.: Extending a brainiac prover to lambda-free higher-order logic. Int. J. Softw. Tools Technol. Transf. **24**(1), 67–87 (2022)
19. Vukmirović, P., Blanchette, J., Schulz, S.: Extending a high-performance prover to higher-order logic. In: Sharygina, N., Sankaranarayanan, S. (eds.) TACAS 2023. LNCS, Springer Cham (2023). https://doi.org/10.1007/978-3-031-30820-8_10

Nominal Techniques for Software Specification and Verification

Maribel Fernández

Department of Informatics, King's College London, London, UK
Maribel.Fernandez@kcl.ac.uk

Abstract. In this talk we discuss the nominal approach to the specification of languages with binders and some applications to programming languages and verification.

Keywords: Binding Operator · Nominal Logic · Nominal Rewriting · Unification · Equational Axioms · Type Systems

Overview

The nominal approach to the specification of languages with binding operators, introduced by Gabbay and Pitts [20, 21, 28], has its roots in nominal set theory [27]. Its user-friendly syntax and first-order presentation (indeed, nominal logic [25, 26] is defined as a theory in first-order logic) makes formal reasoning about binding operators similar to conventional on-paper reasoning.

Nominal logic uses the well-understood concept of *permutation groups acting on sets* to provide a rigorous, first-order treatment of common informal practice to do with fresh and bound names. Nominal matching and nominal unification [36, 37] (which work modulo α-equivalence) are decidable and efficient algorithms exist [7, 8, 9, 22], which are the basis for efficient implementations of nominal rewriting [17–19, 34].

A number of systems (such as Nominal Isabelle [35]) highlighted the benefits of the nominal approach, which gave rise to elegant formalisations of Gödel's theorems [24] and the π-calculus [5] and to advances in programming language semantics [23]. However, there are still some obstacles to the inclusion of nominal features in programming languages and verification environments.

In this talk, I will present our current work towards incorporating nominal techniques into two widely-used rule-based first-order verification environments: the K specification framework [30] and the Maude programming language [11, 12].

An important component of rule-based programming and verification environments is the algorithm used to check equivalence of terms and to solve equations (unification). In practice, unification problems arise in the context of equational axioms (e.g., to take into account associative and commutative (AC) operators [6, 13, 14, 32, 33]). The first part of the talk will discuss notions of α-equivalence modulo associativity and commutativity

Partially funded by the Royal Society (International Exchanges, grant number IES\R2\212106)

axioms [1], extensions of nominal matching and unification to deal with AC operators [2], and the use of nominal narrowing [3] to deal with equational theories presented by convergent nominal rewriting rules.

Another important component of rule-based programming and verification environments is the type system. In the second part of the talk, I will discuss type systems for nominal languages (including polymorphic systems [15] and intersection systems [4]). Dependent type theories, the dominant approach to formalising programming languages, have been extended with nominal features [10, 29, 31]. A lambda-less nominal dependent type system is available [16] and we are currently working on a type checker for this system.

The talk is structured as follows: we will start with the definition of nominal logic (including the notions of fresh atoms and alpha-equivalence) followed by a brief introduction to nominal matching and unification. We will then define nominal rewriting, a generalisation of first-order rewriting that provides in-built support for alpha-equivalence following the nominal approach. Finally, we will discuss notions of nominal unification and rewriting modulo AC operators and briefly overview typed versions of nominal languages.

Acknowledgements. I am grateful to my PhD students and co-authors for many fruitful collaborations.

References

1. Ayala-Rincón, M., de Carvalho Segundo, W., Fernández, M., Nantes-Sobrinho, D., Oliveira, A.: A formalisation of nominal α-equivalence with A, C, and AC symbols. Theor. Comput. Sci. **781**, 3–23 (2019). https://doi.org/10.1016/j.tcs.2019.02.020
2. Ayala-Rincón, M., de Carvalho Segundo, W., Fernández, M., Silva, G.F., Nantes-Sobrinho, D.: Formalising nominal C-unification generalised with protected variables. Math. Struct. Comput. Sci. **31**(3), 286–311 (2021). https://doi.org/10.1017/S0960129521000050
3. Ayala-Rincón, M., Fernández, M., Nantes-Sobrinho, D.: Nominal narrowing. In: Kesner, D., Pientka, B. (eds.) 1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22–26, 2016, Porto, Portugal. LIPIcs, vol. 52, pp. 11:1–11:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.FSCD.2016.11
4. Ayala-Rincón, M., Fernández, M., Rocha-Oliveira, A.C., Ventura, D.L.: Nominal essential intersection types. Theor. Comput. Sci. **737**, 62–80 (2018). https://doi.org/10.1016/j.tcs.2018.05.008
5. Bengtson, J., Parrow, J.: Formalising the pi-calculus using nominal logic. Logical Methods Comput. Sci. **5**(2) (2009), http://arxiv.org/abs/0809.3960
6. Boudet, A., Contejean, E., Devie, H.: A new AC unification algorithm with an algorithm for solving systems of diophantine equations. In: Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990. pp. 289–299. IEEE Computer Society (1990)
7. Calvès, C., Fernández, M.: A polynomial nominal unification algorithm. Theor. Comput. Sci. **403**, 285–306 (2008)

8. Calvès, C., Fernández, M.: Matching and alpha-equivalence check for nominal terms. J. Comput. Syst. Sci. **76**(5), 283–301 (2010)
9. Calvès, C., Fernández, M.: The first-order nominal link. In: Logic-Based Program Synthesis and Transformation - 20th International Symposium, LOPSTR 2010, Hagenberg, Austria, July 23-25, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6564, pp. 234–248. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20551-4_15
10. Cheney, J.: A dependent nominal type theory. Logical Methods Comput. Sci. **8**(1) (2012)
11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, Lecture Notes in Computer Science, vol. 4350. Springer, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
12. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.L.: Programming and symbolic computation in Maude. J. Logic and Algebraic Methods in Programming **110** (2020)
13. Fages, F.: Associative-Commutative Unification. In: Shostak, R.E. (ed.) 7th International Conference on Automated Deduction, Napa, California, USA, May 14–16, 1984, Proceedings. Lecture Notes in Computer Science, vol. 170, pp. 194–208. Springer (1984). https://doi.org/10.1007/978-0-387-34768-4_12
14. Fages, F.: Associative-Commutative Unification. J. of Symbolic Computation **3**(3), 257–275 (1987)
15. Fairweather, E., Fernández, M.: Typed nominal rewriting. ACM Transactions on Computational Logic **19**(1), 6:1–6:46 (2018). https://doi.org/10.1145/3161558
16. Fairweather, E., Fernández, M., Szasz, N., Tasistro, A.: Dependent types for nominal terms with atom substitutions. In: Typed Lambda Calculus and Applications (Proceedings of TLCA). pp. 180–195 (2015). https://doi.org/10.4230/LIPIcs.TLCA.2015.180
17. Fernández, M., Gabbay, M., Mackie, I.: Nominal rewriting systems. In: Moggi, E., Warren, D.S. (eds.) Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 24-26 August 2004, Verona, Italy. pp. 108–119. ACM (2004). https://doi.org/10.1145/1013963.1013978
18. Fernández, M., Gabbay, M.J.: Nominal rewriting. Information and Computation **205**(6), 917–965 (2007)
19. Fernández, M., Gabbay, M.J.: Closed nominal rewriting and efficiently computable nominal algebra equality. In: Crary, K., Miculan,M. (eds.) Proceedings 5th InternationalWorkshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP 2010, Edinburgh, UK, 14th July 2010. EPTCS, vol. 34, pp. 37–51 (2010). https://doi.org/10.4204/EPTCS.34.5
20. Gabbay, M.J.: Foundations of nominal techniques: logic and semantics of variables in abstract syntax. Bulletin of Symbolic Logic (2011)
21. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax with variable binding. Formal Aspects Comput. **13**(3–5), 341–363 (2001)
22. Levy, J., Villaret, M.: An efficient nominal unification algorithm. In: Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010). Leibniz International Proceedings in Informatics (LIPIcs), vol. 6, pp. 209–226. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2010)

23. Murawski, A.S., Tzevelekos, N.: Nominal game semantics. FTPL **2:4**, 191–269 (2016). https://doi.org/10.1561/2500000017
24. Paulson, L.C.: A mechanised proof of Gödel's incompleteness theorems using Nominal Isabelle. J. of Automated Reasoning **55**(1), 1–37 (2015). https://doi.org/10.1007/s10817-015-9322-8
25. Pitts, A.M.: Nominal logic: A first order theory of names and binding. In: Kobayashi, N., Pierce, B.C. (eds.) Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2215, pp. 219–242. Springer (2001). https://doi.org/10.1007/3-540-45500-0_11
26. Pitts, A.M.: Nominal logic, a first order theory of names and binding. Inf. Comput. **186**(2), 165–193 (2003)
27. Pitts, A.M.: Nominal sets: Names and symmetry in computer science. Cambridge UP (2013)
28. Pitts, A.M., Gabbay, M.J.: A metalanguage for programming with bound names modulo renaming. In: Proceedings of the 5th international conference on the mathematics of program construction (MPC 2000). LNCS, vol. 1837, pp. 230–255. Springer (2000)
29. Pitts, A.M., Matthiesen, J., Derikx, J.: A dependent type theory with abstractable names. Electron. Notes Theor. Comput. Sci. **312**, 19–50 (2015). https://doi.org/10.1016/j.entcs.2015.04.003
30. Rosu, G., Serbanuta, T.: An overview of the K semantic framework. J. Log. Algebr. Program. **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012a
31. Schöpp, U., Stark, I.: A Dependent Type Theory with Names and Binding. In: CSL. pp. 235–249 (2004)
32. Stickel, M.: A Unification Algorithm for Associative-Commutative Functions. J. ACM **28**(3), 423–434 (1981)
33. Stickel, M.E.: A Complete Unification Algorithm for Associative-Commutative Functions. In: Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 3–8, 1975. pp. 71–76 (1975)
34. Suzuki, T., Kikuchi, K., Aoto, T., Toyama, Y.: Confluence of Orthogonal Nominal Rewriting Systems Revisited. In: Fernández, M. (ed.) 26th International Conference on Rewriting Techniques and Applications (RTA 2015). Leibniz International Proceedings in Informatics (LIPIcs), vol. 36, pp. 301–317. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015). https://doi.org/10.4230/LIPIcs.RTA.2015.301, http://drops.dagstuhl.de/opus/volltexte/2015/5204
35. Urban, C.: Nominal techniques in Isabelle/HOL. J. Autom. Reason. **40**(4), 327–356 (2008)
36. Urban, C., Pitts, A.M., Gabbay, M.: Nominal unificaiton. In: Baaz, M., Makowsky, J.A. (eds.) Computer Science Logic, 17th International Workshop, CSL 2003, 12th Annual Conference of the EACSL, and 8th Kurt Gödel Colloquium, KGC 2003, Vienna, Austria, August 25-30, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2803, pp. 513–527. Springer (2003). https://doi.org/10.1007/978-3-540-45220-1_41
37. Urban, C., Pitts, A.M., Gabbay, M.J.: Nominal unification. Theor. Comput. Sci. **323**(1–3), 473–497 (2004)

How Can We Trust AI?

Mateja Jamnik

Department of Computer Science and Technology, University of Cambridge, UK
mateja.jamnik@cl.cam.ac.uk

Abstract. Not too long ago most headlines talked about our fear of AI. Today, AI is ubiquitous, and the conversation has moved on from whether we should use AI to how we can trust the AI systems that we use in our daily lives. In this talk I look at some key technical ingredients that help us build confidence and trust in using intelligent technology. I argue that intuitiveness, interaction, explainability and inclusion of human domain knowledge are essential in building this trust. I present some of the techniques and methods we are building for making AI systems that think and interact with humans in more intuitive and personalised ways, enabling humans to better understand the solutions produced by machines, and enabling machines to incorporate human domain knowledge in their reasoning and learning processes.

Keywords: Human-like Computing · Artificial Intelligence · Knowledge Representation · Machine Learning · Automated Reasoning · Cognitive Science.

Over the years the work in this talk has been supported by the EPSRC, Leverhulme Trust, ERC, NSF, Cambridge Gates Foundation, Cambridge Trust.

Contents

Certified Core-Guided MaxSAT Solving

Jeremias Berg[1], Bart Bogaerts[2], Jakob Nordström[3,4], Andy Oertel[3,4](✉), and Dieter Vandesande[2]

[1] HIIT, Department of Computer Science, University of Helsinki, Helsinki, Finland

[2] Artificial Intelligence Laboratory, Vrije Universiteit Brussel, Brussels, Belgium
[3] University of Copenhagen, Copenhagen, Denmark
[4] Lund University, Lund, Sweden
andy.oertel@cs.lth.se

Abstract. In the last couple of decades, developments in SAT-based optimization have led to highly efficient maximum satisfiability (MaxSAT) solvers, but in contrast to the SAT solvers on which MaxSAT solving rests, there has been little parallel development of techniques to prove the correctness of MaxSAT results. We show how pseudo-Boolean proof logging can be used to certify state-of-the-art core-guided MaxSAT solving, including advanced techniques like structure sharing, weight-aware core extraction and hardening. Our experimental evaluation demonstrates that this approach is viable in practice. We are hopeful that this is the first step towards general proof logging techniques for MaxSAT solvers.

Keywords: MaxSAT · core-guided search · proof logging · certifying algorithms

1 Introduction

Combinatorial optimization is one of the most impressive, and most intriguing, success stories in computer science. This area deals with computationally very challenging problems, which are widely believed to require exponential time in the worst case [21,49]. In spite of this, during the last couple of decades astonishing progress has been made on so-called combinatorial solvers for a number of different algorithmic paradigms such as Boolean satisfiability (SAT) solving and optimization [15], constraint programming (CP) [72], and mixed integer programming (MIP) [1,16]. Today, such solvers are routinely used to solve real-world problems with hundreds of thousands or even millions of variables.

While the performance of modern combinatorial solvers is truly impressive, one negative aspect is that they are highly complex pieces of software, and it is well documented that even mature state-of-the-art solvers sometimes give wrong results [2,18,25,37]. This can be fatal for applications where correctness is a non-negotiable demand. Perhaps the most successful approach for addressing this problem so far is the requirement in the SAT solving community that solvers

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 1–22, 2023.
https://doi.org/10.1007/978-3-031-38499-8_1

should be *certifying* [3,62], meaning that when given a formula a solver should output not only a verdict whether the formula is satisfiable or unsatisfiable, but also an efficiently machine-verifiable *proof log* establishing that this verdict is guaranteed to be correct. One can then feed the input formula, the verdict, and the proof log to a special, dedicated *proof checker*, and accept the result if the proof checker agrees that the proof log shows that the solver computation is valid. Over the years, different proof formats such as *RUP* [43], *TraceCheck* [14], *DRAT* [44,45], *GRIT* [27], and *LRAT* [26] have been developed, and for almost a decade *DRAT* proof logging has been compulsory in the (main track of the) SAT competition. However, there has been very limited progress in designing analogous proof logging techniques for more powerful algorithmic paradigms.

Our focus in this work is on the optimization paradigm that is arguably closest to SAT solving, namely *maximum satisfiability* or *MaxSAT* solving [8,56], and the challenge of developing proof logging techniques for MaxSAT solvers.

1.1 Previous Work

Since essentially all modern MaxSAT solvers are based on repeated invocations of SAT solvers, a first question is why SAT proof logging techniques are not sufficient. While *DRAT* is a very powerful proof system, it seems that the overhead of generating proofs of correctness for the rewriting steps in between SAT solver calls in MaxSAT solvers is too large to be tolerable for practical purposes. Another, related, problem is that for optimization problems one needs to reason about the objective function, which *DRAT* struggles to do since its language is limited to disjunctive clauses. But perhaps the biggest challenge is that while modern SAT solving is completely dominated by the *conflict-driven clause learning (CDCL)* method [11,59,66], for MaxSAT there is a rich variety of approaches including *linear SAT-UNSAT* (or *model-improving search*) [31,54,68], *core-guided search* [4,7,35,67], *implicit hitting set (IHS)* search [28,29], and some recent work on branch-and-bound methods [57] (where we stress that the lists of references are far from exhaustive).

One tempting solution to circumvent this heterogeneity of solving approaches is to treat the MaxSAT solver as a black box and use a single call to a certifying SAT solver to prove optimality of the final solution found. However, there are several problems with this proposal. Firstly, we would still need proof logging to ensure that the input to the SAT solver is a correct encoding of a claim of optimality for the correct problem instance. Secondly, such a SAT call could be extremely expensive, running counter to the goal of proof logging with low (and predictable) overhead. Finally, even if the SAT-call approach could be made to work efficiently, this would just certify the final result, and would not help validate the correctness of the reasoning of the solver. For these reasons, our goal is to provide proof logging for the actual computations of the MaxSAT algorithm.

While some proof systems and tools have been developed specifically for MaxSAT [19,34,48,53,64,65,69–71], none of them comes close to providing general-purpose proof logging, because they apply only for very specific algorithm implementations and/or fail to capture the full range of reasoning used in an algorithmic approach. A recent work [75] by two co-authors on the current paper instead leverages the pseudo-Boolean proof logging system VeriPB [76] to certify correctness of the unweighted linear SAT-UNSAT solver QMaxSAT. VeriPB is similar in spirit to *DRAT*, but operates with more general 0–1 linear inequalities rather than just clauses. This simplifies reasoning about optimization problems, and also makes it possible to capture the powerful MaxSAT solver inferences in a more concise way. VeriPB has previously been used for proof logging of enhanced SAT solving techniques [17,42] and pseudo-Boolean solving [38], as well as for providing proof-of-concept tools for a nontrivial range of techniques in constraint programming [33,41] and subgraph solving [39,40].

1.2 Our Contributions

In this work, we use VeriPB to provide, to the best of our knowledge for the first time, efficient proof logging for the full range of techniques in a cutting-edge MaxSAT solver. We consider the state-of-the-art core-guided solver CGSS [47], based on RC2 [46], and show how to enhance CGSS to output proofs of correctness of its reasoning, including sophisticated techniques such as stratification [6,58], intrinsic-at-most-one constraints [46], hardening [6], weight-aware core-extraction [13], and structure sharing [47]. We find that the overhead for such proof logging is perfectly manageable, and although there is certainly room to improve the proof verification time, our experiments demonstrate that already a first proof-of-concept implementation of this approach is practically feasible.

It has been shown previously [32,39,52] that proof logging can also serve as a powerful debugging tool. This is because faulty reasoning is likely to lead to unsound proofs, which can be detected even if the solver produces correct output for all test cases. We exhibit yet another example of this—some proofs for which we struggled to make the verification work turned out to reveal two well-hidden bugs in RC2 and CGSS that earlier extensive testing had failed to uncover.

Although it still remains to provide proof logging for other MaxSAT approaches such as (general, weighted) linear SAT-UNSAT and implicit hitting set (IHS) search, we are optimistic that our work could serve as an important step towards general adoption of proof logging techniques for MaxSAT solvers.

1.3 Outline of This Paper

After reviewing preliminaries for pseudo-Boolean reasoning and core-guided MaxSAT solving in Sects. 2 and 3, we explain how core-guided MaxSAT solvers can be equipped with proof logging methods in Sect. 4. In Sect. 5 we present our experimental evaluation, after which some concluding remarks and directions for future research are given in Sect. 6.

2 Preliminaries

We start by a review of some standard material which can be found, e.g., in [20, 38,42]. A *literal* ℓ over a Boolean variable x (taking values in $\{0,1\}$, which we identify with false and true, respectively) is x itself or its negation $\overline{x}$, where $\overline{x} = 1 - x$. A *pseudo-Boolean (PB)* constraint is a 0-1 integer linear inequality $C \doteq \sum_i a_i \ell_i \geq A$ (where $\doteq$ denotes syntactic equality). When convenient, we can assume without loss of generality that PB constraints are in *normalized form* [10]; i.e., all literals ℓ_i are over distinct variables and the coefficients a_i and the *degree (of falsity)* A are non-negative integers. The set of literals in C is denoted *lits*(C). The *negation* of C is $\neg C \doteq \sum_i a_i \ell_i \leq A - 1$ (rewritten in normalized form when needed). A *pseudo-Boolean formula* is a conjunction $F \doteq \bigwedge_j C_j$ of PB constraints. Note that a disjunctive clause can be viewed as a PB constraint with all coefficients and the degree equal to 1, and so formulas in conjunctive normal form (CNF) are special cases of PB formulas.

A *(partial) assignment* ρ is a (partial) function from variables to $\{0, 1\}$, which we extend to literals by respecting the meaning of negation. Applying ρ to a constraint C yields $C\restriction_\rho$ by substituting the variables assigned in ρ by their values, and for a formula $F \doteq \bigwedge_j C_j$ we define $F\restriction_\rho \doteq \bigwedge_j C_j\restriction_\rho$. The constraint C is *satisfied* by ρ if $\sum_{\rho(\ell_i)=1} a_i \geq A$, and ρ satisfies F if it satisfies all $C \in F$, in which case F is *satisfiable*. A formula lacking satisfying assignments is *unsatisfiable*. We say that F *implies* C, denoted $F \models C$, if any assignment satisfying F also satisfies C.

An *objective* $O \doteq \sum_i w_i \ell_i + M$ is an affine function over literals ℓ_i to be minimized by (total) assignments α satisfying F. The *value* (or *cost*) of an objective O under such an α, which we refer to as a *solution*, is $O(\alpha) = \sum_{\alpha(\ell_i)=1} w_i + M$. We write *coeff*$(O, \ell_i)$ to denote the coefficient w_i of a literal $\ell_i \in$ *lits*(O).

The foundation of the pseudo-Boolean proof logging in this paper is the *cutting planes* proof system [24], which is a method to iteratively derive new constraints implied by a pseudo-Boolean formula F. If C and D have been derived before or are *axiom constraints* in F, then any positive *linear combination* of these constraints can be derived. *Literal axioms* $\ell \geq 0$ can also be added to any previously derived constraints. For a constraint $\sum_i a_i \ell_i \geq A$ in normalized form, *division* by a positive integer d derives $\sum_i \lceil a_i/d \rceil \ell_i \geq \lceil A/d \rceil$, and we also add a *saturation* rule that derives $\sum_i \min\{a_i, A\} \cdot \ell_i \geq A$ (where the soundness of these rules crucially depends on the normalized form). It is well known that any PB constraint implied by F can be derived using these rules.

A constraint C is said to *unit propagate* the literal ℓ to true under an assignment ρ if $C\restriction_\rho$ cannot be satisfied unless ℓ is true. During *unit propagation* on F under ρ, we extend ρ iteratively by any propagated literals until an assignment ρ' is reached under which no constraint $C \in F$ is propagating or some constraint C wants to propagate a literal that has already been assigned to the opposite value. The latter case is called a *conflict*, since C is *violated* by ρ'. We say that F implies C by *reverse unit propagation (RUP)*, and that C is a *RUP constraint* with respect to F, if $F \wedge \neg C$ unit propagates to conflict under the empty assignment. It is not hard to see that $F \models C$ holds if C is a RUP constraint, and as a convenient shorthand we will add a RUP rule for deriving new constraints.

In addition to deriving constraints that are implied by a formula F, we also allow deriving so-called *redundant* constraints C that are *not* implied by F as long as some optimal solution is guaranteed to be preserved. This is done by extending the proof system with a *redundance-based strengthening* rule [17,42]. We will only need the special case of this rule saying that for a fresh variable z and for any constraint $D \doteq \sum_i a_i \ell_i \geq A$ we can introduce the *reified constraints*

$$C^{\Rightarrow}_{\text{reif}}(z, D) \doteq A\overline{z} + \textstyle\sum_i a_i \ell_i \geq A \tag{1a}$$

$$C^{\Leftarrow}_{\text{reif}}(z, D) \doteq \left(\textstyle\sum_i a_i - A + 1\right) z + \textstyle\sum_i a_i \overline{\ell}_i \geq \textstyle\sum_i a_i - A + 1 \tag{1b}$$

encoding the implications $z \Rightarrow D$ and $z \Leftarrow D$, respectively. We refer to z as the *reification variable*, and when D is clear from context, we will sometimes write just $C^{\Rightarrow}_{\text{reif}}(z)$ for (1a) and $C^{\Leftarrow}_{\text{reif}}(z)$ for (1b).

The *maximum satisfiability (MaxSAT) problem* can be described conveniently as a special case of pseudo-Boolean optimization. A discussion on the equivalence of the following and the—more classical—clause-centric definition can be found in, for instance, [8,55]. An instance (F, O) of the (weighted partial) MaxSAT problem consists of a CNF formula F and an objective function O written as a non-negative affine combination of literals. The goal is to find a solution α that satisfies F and minimizes $O(\alpha)$. We say that such a solution α is *optimal* for the instance and that the optimal cost of the instance (F, O) is $O(\alpha)$.

3 The OLL Algorithm for Core-Guided MaxSAT Solving

We now proceed to discuss the core-guided MaxSAT solving in CGSS, which is based on the OLL algorithm [5,63], and describe the main heuristics used in efficient implementations of this algorithm. Given a MaxSAT instance (F_{orig}, O_{orig}), OLL takes an optimistic view and attempts to find an assignment satisfying F_{orig} in which O_{orig} equals its constant term (i.e., all literals in $\mathit{lits}(O_{orig})$ are false). If such a solution exists, it is clearly optimal. Otherwise, the solver will extract a *core* K, which is a clause such that (i) K only contains objective literals, i.e., $\mathit{lits}(K) \subseteq \mathit{lits}(O_{orig})$, and (ii) F_{orig} implies K, which means that any solution to F_{orig} has to set at least one literal in $\mathit{lits}(K)$ to true. The *cost* $w(K, O) = \min\{\mathit{coeff}(O, \ell) : \ell \in \mathit{lits}(K)\}$ of a core K is the smallest coefficient in the objective O of any literal in K. The core K is used to (conceptually) reformulate the instance into (F_{ref}, O_{ref}) which has the same minimal-cost solutions. The constant term LB in O_{ref} is a lower bound on the optimal cost of the instance, and the reformulation is done in such a way that the lower bound increases (exactly) with the cost of the core K as defined above.

In more detail, the algorithm maintains a reformulated objective O_{ref} (initialized to O_{orig}) such that the (non-normalized) pseudo-Boolean constraint

$$O_{orig} \geq O_{ref} \doteq \sum_{b \in \mathit{lits}(O_{orig})} \mathit{coeff}(O_{orig}, b) \cdot b \geq \sum_{b' \in \mathit{lits}(O_{ref})} \mathit{coeff}(O_{ref}, b') \cdot b' + LB \tag{2}$$

is satisfied by all solutions of F_{ref}. Note that the constraint (2), which we refer to as an *objective reformulation constraint*, implies that the constant term LB is a lower bound on the optimal cost.

In each iteration, a SAT solver is queried for a solution α to F_{ref} with $O_{ref}(\alpha) = LB$. If such an α exists, the constraint (2) yields that $O_{orig}(\alpha) = LB$, and so α is a minimal-cost solution to (F_{orig}, O_{orig}). Otherwise, the solver returns a new core K that requires at least one literal in $\mathit{lits}(O_{ref})$ to be set to 1. This implies that the optimal cost is strictly larger than LB, and the core K is used for a new reformulation step.

The objective reformulation step adds new clauses to F_{ref} encoding the constraints $y_{K,k} \Leftarrow \sum_{b \in Lit(K)} b \geq k$ for $k = 2, \ldots, |K|$. The new variables $y_{K,k}$ are added to O_{ref} with coefficient $w(K, O_{ref})$ equalling the cost of K, and the coefficient in O_{ref} of each literal in K is decreased by the same amount. Finally, the lower bound LB—the constant term of O_{ref}—is also increased by $w(K, O_{ref})$. Since $y_{K,k}$ encodes that at least k literals in K are true, we have the equality $\sum_{b \in \mathit{lits}(K)} b = 1 + \sum_{k=2}^{|K|} y_{K,k}$, where the additive 1 comes from the fact that at least one literal in K has to be true, and the reformulation step is just applying this equality multiplied by $w(K, O_{ref})$ to O_{ref}. Notice that the variables added during objective reformulation can later be discovered in other cores. In practice, all implementations of OLL we are aware of encode the semantics of counting variables incrementally [60]. This means that initially only the variable $y_{K,2}$ is defined, and the variable $y_{K,i+1}$ is introduced only after $y_{K,i}$ is found in a core.

Implementations of OLL for MaxSAT—including the CGSS solver that we enhance with proof logging in this work—extend the algorithm with a number of heuristics such as stratification [6,58], hardening [6], the intrinsic-at-most-ones technique [46], weight-aware core extraction [13], and structure sharing [47].

Stratification extracts cores not over all literals in O_{ref} but only over those whose coefficient is above some bound w_{strat}. This steers search toward cores containing literals with high coefficients, resulting in larger increases of LB. Once no more cores over such variables can be found, the algorithm lowers w_{strat}, terminating only after no more cores can be found with $w_{strat} = 1$. The fact that no more cores containing only variables with coefficients above w_{strat} exist is detected by the SAT solver returning a (possibly non-optimal) solution α. The minimal cost $O_{orig}(\alpha)$ of all such solutions gives an upper bound UB on the optimal cost of the instance, allowing OLL to terminate as soon as $LB = UB$.

Hardening fixes literals in O_{ref} to 0 based on information provided by the current upper and lower bounds UB and LB. If for any $b \in \mathit{lits}(O_{ref})$ it holds that $\mathit{coeff}(O_{ref}, b) + LB > UB$, then any solution α with $b = 1$ would have higher cost than the current best solution known, and would thus not be optimal.

The *intrinsic-at-most-one* technique identifies subsets $\mathcal{S} \subseteq \mathit{lits}(O_{ref})$ of objective literals such that $\sum_{b \in \mathcal{S}} \overline{b} \leq 1$ is implied, i.e., any solution can assign at most one literal in $\mathcal{S}$ to 0. This is used both to increase the lower bound and to reformulate the objective. If we let $w_{min} = \min\{\mathit{coeff}(O_{ref}, b) : b \in \mathcal{S}\}$, then $\mathcal{S}$ implies a lower bound increase of $LB_{\mathcal{S}} = (|\mathcal{S}| - 1) \cdot w_{min}$. Additionally, we define a new variable $\ell_{\mathcal{S}}$ by the clause $\ell_{\mathcal{S}} + \sum_{b \in \mathcal{S}} \overline{b} \geq 1$ to indicate if in fact all literals in $\mathcal{S}$

are true, and introduce it in the reformulated objective with coefficient w_{min}. This means that we remove the already known lower bound $LB_{\mathcal{S}}$ from O_{ref} and transfer the possible additional cost w_{min} from $\mathcal{S}$ to the variable $\ell_{\mathcal{S}}$.

Weight-aware core extraction (WCE) delays objective reformulation, and the accompanying increase in new variables and clauses, for as long as possible. When a new core K is extracted by a solver that uses WCE, initially only the coefficient of each $b \in \mathit{lits}(K)$ is lowered and the lower bound LB is increased by $w(K, O_{ref})$. Then the SAT solver is invoked again with the literals, that still have coefficients above w_{strat} in O_{ref}, set to 0. When the SAT solver finds a satisfying assignment extending the assumptions, all objective reformulations steps are then performed at once. This is correct since the final effect is the same as if the core would have been discovered one by one and immediately followed by objective reformulation. Notice that this core extraction loop is guaranteed to terminate since the coefficient of at least one variable is decreased to 0 for each new core. *Structure sharing* is a recent extension to weight-aware core extraction that makes use of the potential overlap in cores detected in order to achieve more compact encodings of counting variable semantics.

4 Proof Logging for the OLL Algorithm for MaxSAT

We have now reached a point where we can describe the contribution of this work, namely how to add proof logging to an OLL-based core-guided MaxSAT solver, including all the state-of-the-art techniques described in Sect. 3.

In our proof logging routines we maintain the invariants described next. The reformulated objective O_{ref} is already implicitly tracked by the solver and at all times it is possible to derive that $O_{orig} \geq O_{ref}$ as in (2). We also keep track of the current upper bound UB on O_{orig} and best solution α_{best} found so far. All cores that have been found and processed are in the set $\mathcal{K}$.

SAT Solver Calls. The CDCL SAT solvers used in core-guided MaxSAT algorithms can support *DRAT* proof logging, and since the proof format used by VeriPB is a strict extension of *DRAT* (modulo small and purely syntactical modifications) it is straightforward to provide proof logging for the part of the reasoning done in SAT solver calls, and to add all learned clauses to the proof checker database.

Each invocation of the SAT solver returns either a new solution α or a new core K. When a solution α with $O_{orig}(\alpha) < UB$ is obtained, it is logged in the proof, which adds the *objective-improving constraint*

$$O_{orig} \leq UB - 1 \tag{3a}$$

(which is

$$\sum_{b \in \mathit{lits}(O_{orig})} \mathit{coeff}(O_{orig}, b) \cdot \overline{b} \;\geq\; 1 + \sum_{b \in \mathit{lits}(O_{orig})} \mathit{coeff}(O_{orig}, b) - UB \tag{3b}$$

in normalized form). A technical side remark is that later solutions with cost greater than UB cannot successfully be logged, since they violate the constraint (3a) added to the proof checker database, and so the proof logging routines make sure to only log solutions that improve the current upper bound.

If the SAT solver instead returns a new core K, this clause is guaranteed to be a reverse unit propagation (RUP) clause with respect to the set of clauses currently in the solver database, and so we can use the RUP rule to add K to the proof checker database (which contains a superset of the clauses known by the solver). For our book-keeping, we also add K to the set $\mathcal{K}$. A special case is that K could be the contradictory empty clause, corresponding to the pseudo-Boolean constraint $0 \geq 1$. This means that there are no solutions to the formula.

To optimize the efficiency of proof verification, constraints should be deleted from the proof when they are no longer needed. Since SAT solver proofs are only used to prove *unsatisfiability* this does not cause any issues, but when certifying *optimality* we have to be careful in order not to create better-than-optimal solutions (which could happen if, e.g., constraints in the input formula are removed). The *checked deletion* rule [17] ensuring this in VeriPB does not have any analogue in *DRAT*, so some care is needed here when translating SAT solver proofs into the VeriPB format.

Incremental Totalizer with Structure Sharing. Different implementations of OLL for MaxSAT differ in which encoding is used for the counting variables introduced during objective reformulation [9,50,51]. The two solvers we consider use totalizers [9], so we start by explaining this encoding and then show how to provide proof logging for the clauses added to the proof checker database.

The totalizer encoding for a set $I = \{\ell_1, \ldots, \ell_n\}$ of literals is a CNF formula $\mathcal{T}$ that defines *counting variables* $y_{I,j}$ for $j = 1, \ldots, n$ such that for any assignment that satisfies $\mathcal{T}$ the variable $y_{I,j}$ is true if and only if $\sum_{i=1}^{n} \ell_i \geq j$. The structure of $\mathcal{T}$ can be viewed as a binary tree, with literals in I at the leaves and with each internal node η associated with variables counting the true leaf literals in the subtree rooted at η. The variables $y_{I,j}$ are associated with the root of the tree.

More formally, given a set of literals I, we construct a binary tree with leaves labelled by the literals in I. For every node η of $\mathcal{T}$, let $\mathit{lits}(\eta)$ denote the leaves in the subtree rooted at η; where it is convenient, we will overload I to also refer to the root note. For each internal node η, the totalizer encoding introduces the counting variables $S_\eta = \{y_{\eta,1}, \ldots, y_{\eta,|\mathit{lits}(\eta)|}\}$, the meaning of which can be encoded recursively in terms of the variables S_{η_1} and S_{η_2} for the children η_1 and η_2 of η by the (pseudo-Boolean form of the) clauses

$$C_\eta^{\Leftarrow}(\alpha,\beta,\sigma) \doteq y_{\eta,\sigma} + \overline{y}_{\eta_1,\alpha} + \overline{y}_{\eta_2,\beta} \geq 1 \tag{4a}$$

$$C_\eta^{\Rightarrow}(\alpha,\beta,\sigma) \doteq \overline{y}_{\eta,\sigma+1} + y_{\eta_1,\alpha+1} + y_{\eta_2,\beta+1} \geq 1 \tag{4b}$$

for all integers α, β, σ such that $\alpha + \beta = \sigma$ and $0 \leq \alpha \leq |\mathit{lits}(\eta_1)|$, $0 \leq \beta \leq |\mathit{lits}(\eta_2)|$, and $0 \leq \sigma \leq |\mathit{lits}(\eta)|$. We use the notational conventions in (4a)–(4b) that $y_{\ell,1} = \ell$ for all leaves ℓ, and that $y_{\eta,0} = 1$ and $y_{\eta,|\mathit{lits}(\eta)|+1} = 0$ for all nodes η (so that clauses containing $y_{\eta,0}$ or $y_{\eta,|\mathit{lits}(\eta)|+1}$ can be simplified to binary clauses or be omitted when they are satisfied). The clauses $C_\eta^{\Rightarrow}(\alpha,\beta,\sigma)$

in (4b) are not necessarily added to the clause database of the MaxSAT solver, but are sometimes included for improved propagation.

We now turn to the question of how to derive the clauses (4a)–(4b) encoding the meaning of the counting variables $y_{I,j}$ in the proof. This is a two-step process. First, reified pseudo-Boolean (and, in general, non-clausal) constraints $C^{\Rightarrow}_{\text{reif}}(y_{\eta,j})$ and $C^{\Leftarrow}_{\text{reif}}(y_{\eta,j})$ as in (1a)–(1b), encoding that $y_{\eta,j}$ holds if and only if $\sum_{\ell \in lits(\eta)} \ell \geq j$, are derived by redundance-based strengthening. Then the clauses added to the MaxSAT solver are derived from these pseudo-Boolean constraints. Although we omit the details due to space constraints, it is not hard to show that for any internal node η with children η_1 and η_2, the clauses $C^{\Leftarrow}_{\eta}(\alpha, \beta, \sigma)$ and $C^{\Rightarrow}_{\eta}(\alpha, \beta, \sigma)$ in (4a)–(4b) can be derived from the constraints $C^{\Leftarrow}_{\text{reif}}(y_{\eta,\sigma})$, $C^{\Rightarrow}_{\text{reif}}(y_{\eta,\sigma})$, $C^{\Leftarrow}_{\text{reif}}(y_{\eta_1,\alpha})$, $C^{\Rightarrow}_{\text{reif}}(y_{\eta_1,\alpha})$, $C^{\Leftarrow}_{\text{reif}}(y_{\eta_2,\beta})$, and $C^{\Rightarrow}_{\text{reif}}(y_{\eta_2,\beta})$ by standard cutting planes derivations as in [75]. In particular, the certification of these totalizers can be done incrementally: clauses in the encoding can be derived as the corresponding counter variables are lazily introduced in the OLL algorithm.

This approach is also compatible with structure sharing, where subtrees of a previously constructed totalizer tree can be reused (to avoid doing the same work twice). The only constraints from a subtree rooted at η^* that are needed when generating another totalizer encoding at a higher level are the constraints $C^{\Rightarrow}_{\text{reif}}(y_{\eta^*,\sigma})$ and $C^{\Leftarrow}_{\text{reif}}(y_{\eta^*,\sigma})$ defining the counter variables in the subtree root η^*.

To decrease the memory usage of the proof checker, it can be useful to *delete* reification constraints from the proof once we know that they will no longer be needed. Without structure sharing, for an internal node η, once all clauses that mention $y_{\eta,j}$ are created, the constraints $C^{\Leftarrow}_{\text{reif}}(y_{\eta,j})$ and $C^{\Rightarrow}_{\text{reif}}(y_{\eta,j})$ will not be used anymore and can thus be deleted. On the other hand, structure sharing reuses as many counting variables as possible, even over multiple iterations of weight-aware core extraction. This means that $C^{\Leftarrow}_{\text{reif}}(y_{\eta,j})$ and $C^{\Rightarrow}_{\text{reif}}(y_{\eta,j})$ need to be retained, even after all clauses in the totalizer encoding for all parents of node η have been created.

Objective Reformulation. If counting variables $y_{K,i}$ for $i = 2, \ldots, s_K$ have been introduced for the core K, then the objective reformulation with respect to K is derived with the help of the constraint

$$\sum_{b \in K} b \geq 1 + \sum_{i=2}^{s_K} y_{K,i} \tag{5a}$$

(or

$$\sum_{b \in K} b + \sum_{i=2}^{s_K} \overline{y}_{K,i} \geq s_K \tag{5b}$$

in normalized form). The constraint (5b) can in turn be obtained from the core clause K and the reified constraints $C^{\Rightarrow}_{\text{reif}}(y_{K,j})$. It is clear that this should be possible, since the latter constraints define the variables $y_{K,j}$ precisely so that (5b) should hold, and we refer to Algorithm 5 in [38] for the details. Also, each time a new counting variable $y_{K,j}$ is introduced for a core K, we add it to (5b) to maintain this constraint as an invariant.

To illustrate how this update works, suppose we have a core $K \doteq \sum_{i=1}^{n} b_i \geq 1$ for which $\sum_{i=1}^{n} b + \sum_{i=2}^{s_K - 1} \overline{y}_{K,i} \geq s_K - 1$ has already been derived. The next counting variable y_{K,s_K} is introduced by the reification $s_K \cdot \overline{y}_{K,s_K} + \sum_{i=1}^{n} b_i \geq s_K$. The previous constraint is multiplied by $s_K - 1$ and added to the new reified constraint, yielding $s_K \cdot \sum_{i=1}^{n} b + (s_K - 1) \cdot \sum_{i=2}^{s_K - 1} \overline{y}_{K,i} + s_K \cdot \overline{y}_{K,s_K} \geq (s_K - 1) \cdot s_K + 1$. Dividing this last constraint by s_K results in $\sum_{i=1}^{n} b + \sum_{i=2}^{s_K} \overline{y}_{K,i} \geq s_K$, which is the desired updated constraint.

For a set of extracted cores $\mathcal{K}$, we can derive the objective reformulation constraint $O_{orig} \geq O_{ref}$ by multiplying (5b) for each $K \in \mathcal{K}$ by the cost $w(K, O_{ref})$ of K and summing up all these multiplied constraints. The fact that we have an inequality $O_{orig} \geq O_{ref}$ rather than an equality is due to the incremental use of totalizers. More specifically, if $s_K = |\mathit{lits}(K)|$ would hold for every $K \in \mathcal{K}$, it would be possible to derive $O_{orig} = O_{ref}$ instead. Here we would like to stress one subtlety for developing proof logging for OLL: as the algorithm progresses and more output variables of totalizers are introduced (i.e., the counters s_K increase), the reformulated objective potentially also increases—because of added counted variables when s_K increases we have the inequality $O_{orig} \geq O_{ref}^{new} \geq O_{ref}^{old}$. For this reason, the old constraint $O_{orig} \geq O_{ref}^{old}$ cannot be used to derive $O_{orig} \geq O_{ref}^{new}$ after objective reformulation. Instead, we have to derive $O_{orig} \geq O_{ref}$ from scratch each time the solver argues with the reformulated objective. For doing this we need to have access to the entire set $\mathcal{K}$ of cores.

Proving Optimality. When the solver has found an optimal solution and established a matching lower bound, optimality is certified in the proof log using a proof by contradiction from the objective reformulation constraint $O_{orig} \geq O_{ref}$ in (2) and the (normalized form of the) objective-improving constraint $O_{orig} \leq UB - 1$ in (3b). If we add these two constraints and cancel like terms, we get

$$\sum_{b' \in \mathit{lits}(O_{ref})} \mathit{coeff}(O_{ref}, b') \cdot \overline{b}' \geq 1 - UB + LB + \sum_{b' \in \mathit{lits}(O_{ref})} \mathit{coeff}(O_{ref}, b') \,. \qquad (6)$$

Since we have $UB = LB$ when the optimal solution has been found, and since $\sum_{b' \in \mathit{lits}(O_{ref})} \mathit{coeff}(O_{ref}, b') \cdot \overline{b}'$ cannot possibly exceed $\sum_{b' \in \mathit{lits}(O_{ref})} \mathit{coeff}(O_{ref}, b')$, the constraint (6) can be simplified to contradiction $0 \geq 1$.

Intrinsic At-Most-One Constraints. Certifying intrinsic at-most-one constraints for a set $\mathcal{S} \subseteq \mathit{lits}(O_{ref})$ of literals requires deriving (i) the at-most-one constraint stating that at most one $b \in \mathcal{S}$ is assigned to 0 by any solution and (ii) constraints defining the variable $\ell_{\mathcal{S}}$. Such sets $\mathcal{S}$ are detected by unit propagation that implicitly derives implications $\overline{b}_i \Rightarrow b_j$ in the form of binary clauses $b_i + b_j \geq 1$ for every pair of variables in $\mathcal{S}$. In the proof log, all these binary clauses can be obtained by RUP steps, after which the at-most-one constraint $\sum_{b \in \mathcal{S}} \overline{b} \leq 1$ (which is $\sum_{b \in \mathcal{S}} b \geq |\mathcal{S}| - 1$ in normalized form) is derived by a standard cutting planes derivation (see, e.g., [24]).

The reified constraints $\ell_{\mathcal{S}} \Leftarrow \sum_{b \in \mathcal{S}} b \geq |S|$ and $\ell_{\mathcal{S}} \Rightarrow \sum_{b \in \mathcal{S}} b \geq |S|$ defining the variable $\ell_{\mathcal{S}}$ (which are $\ell_{\mathcal{S}} + \sum_{b \in \mathcal{S}} \overline{b} \geq 1$ and $\overline{\ell}_{\mathcal{S}} + \sum_{b \in \mathcal{S}} b \geq |\mathcal{S}|$, respectively, in normalized form) are derived by redundance-based strengthening. Note that the latter constraint does not exist in the MaxSAT solver, but we need it in the proof in order to derive the objective reformulation for the at-most-one constraint.

Hardening. Formally, hardening corresponds to deriving $\overline{b} \geq 1$ in the proof for some literal $b \in \mathit{lits}(O_{\mathit{ref}})$ for which $\mathit{UB} < \mathit{LB} + \mathit{coeff}(O_{\mathit{ref}}, b)$ holds. Such an inequality $\overline{b} \geq 1$ is implied by RUP if we first derive the constraint (6), since assigning $b = 1$ results in (6) being contradicting.

Upper Bound Estimation. A final technical proof logging detail is that some implementations of the OLL algorithm for MaxSAT—including the Python-based version of CGSS—do not use the actual cost of the solution found by the SAT solver as the upper bound UB when hardening. In order to avoid the overhead in Python of extracting the solution from the SAT solver, an upper bound estimate $\mathit{UB}_{\mathit{est}}$ is computed instead based on the initial assignment passed to the SAT solver in the call. Since any valid estimate is at least the cost of the solution found (i.e., $\mathit{UB}_{\mathit{est}} \geq \mathit{UB}$), hardening steps based on $\mathit{UB}_{\mathit{est}}$ can be justified by first deriving $O_{\mathit{orig}} \leq \mathit{UB}_{\mathit{est}} - 1$, which follows from the latest objective-improving constraint (3a). However, in order to handle solutions correctly in the proof, the proof logging routines need to extract the solution found by the solver and compute the actual cost, which means that a Python-based solver will not be able to avoid this overhead when running with proof logging.

Worked-Out Example. We end this section with a complete, worked-out example of OLL solving and proof logging for the toy MaxSAT instance (F, O) with formula $F = \{(b_1 \vee x), (\neg x \vee b_2), (b_3 \vee b_4)\}$ and objective $O = 5b_1 + 5b_2 + b_3 + b_4$.

After initialization, the internal SAT solver of the OLL algorithm is loaded with the clauses of F and the proof consists of constraints (1)–(3) in Table 1. The OLL search begins by invoking the SAT solver on the clauses in F in order to check the existence of any solutions. Assume the SAT solver returns the solution α_1 assigning $b_1 = b_3 = b_4 = 1$ and $b_2 = x = 0$. This solution has objective value $O(\alpha_1) = O_{\mathit{orig}}(\alpha_1) = 7$ so the algorithm updates $\mathit{UB} = 7$ and logs the objective-improving constraint (4) in Table 1 equivalent to $O_{\mathit{orig}} \leq 6$.

Assume the stratification bound w_{strat} is initialised to 2. Then the solver is invoked with $b_1 = b_2 = 0$ and returns the core $K_1 \doteq b_1 + b_2 \geq 1$, which is added to the proof as constraint (5). As already mentioned, core clauses are guaranteed to be RUP with respect to the set of clauses in the SAT solver database, which are also added to the proof.

For simplicity, we ignore WCE and structure sharing in this example, meaning that the solver next reformulates the objective based on K_1 by introducing clauses enforcing $y_{K_1,2} \Leftarrow (b_1 + b_2 \geq 2)$ for the new counting variable $y_{K_1,2}$. This is done by (i) introducing the pseudo-Boolean constraints (6) and (7) in Table 1 by reification, and (ii) deriving the clauses corresponding to these constraints. While the MaxSAT solver only uses the implication (6), the proof also requires

Table 1. Example proof produced by a certified OLL solver.

id	Pseudo-Boolean constraint	Justification
(1)	$b_1 + x \geq 1$	input
(2)	$b_2 + \overline{x} \geq 1$	input
(3)	$b_3 + b_4 \geq 1$	input
(4)	$5\overline{b}_1 + 5\overline{b}_2 + \overline{b}_3 + \overline{b}_4 \geq 6$	log solution α_1
(5)	$b_1 + b_2 \geq 1$	RUP
(6)	$\overline{b}_1 + \overline{b}_2 + y_{K_1,2} \geq 1$	reification
(7)	$2\overline{y}_{K_1,2} + b_1 + b_2 \geq 2$	reification
(8)	$5b_1 + 5b_2 + 5\overline{y}_{K_1,2} \geq 10$	$(((5) + (7))/2) \cdot 5$
(9)	$\overline{b}_3 + \overline{b}_4 + 5\overline{y}_{K_1,2} \geq 6$	$(4) + (8)$
(10)	$\overline{y}_{K_1,2} \geq 1$	RUP
(11)	$b_3 + b_4 \geq 1$	RUP
(12)	$\overline{b}_3 + \overline{b}_4 + y_{K_2,2} \geq 1$	reification
(13)	$2\overline{y}_{K_2,2} + b_3 + b_4 \geq 2$	reification
(14)	$b_3 + b_4 + \overline{y}_{K_2,2} \geq 2$	$((11) + (13))/2$
(15)	$5\overline{b}_1 + 5\overline{b}_2 + \overline{b}_3 + \overline{b}_4 \geq 7$	log solution α_2
(16)	$5b_1+5b_2+b_3+b_4+5\overline{y}_{K_1,2}+\overline{y}_{K_2,2} \geq 12$	$(8) + (14)$
(17)	$5\overline{y}_{K_1,2} + \overline{y}_{K_2,2} \geq 7$	$(15) + (16)$, $\perp$

constraint (7) corresponding to $y_{K_1,2} \Rightarrow (b_1 + b_2 \geq 2)$. Conveniently, in this toy example $y_{K_1,2} \Leftarrow (b_1 + b_2 \geq 2)$ is already the clause $\overline{b}_1 + \overline{b}_2 + y_{K_1,2} \geq 1$, so step (ii) is not needed. For the general case, we derive totalizer clauses as explained in Sect. 4. Conceptually, we now replace $5b_1 + 5b_2$ by $5y_{K_1,2} + 5$ to obtain the reformulated objective $O_{ref} = b_3 + b_3 + 5y_{K_1,2} + 5$ with lower bound $LB = 5$. The core K_1 says that at least one of b_1 and b_2 must be true, thus incurring a cost of 5, and $y_{K_1,2}$ is added to the objective to indicate if both of them incur cost.

Since it now holds that $\mathit{coeff}(O_{ref}, y_{K_1,2}) + LB = 5 + 5 \geq 7 = UB$, the literal $y_{K_1,2}$ is hardened to 0. In order to certify this hardening step, i.e., derive $\overline{y}_{K_1,2} \geq 1$, the proof logger first derives the objective reformulation constraint $5b_1 + 5b_2 + b_3 + b_4 \geq b_3 + b_4 + 5y_{K_1,2} + 5$ enforced by line (8) in Table 1. The objective-improving and objective reformulation constraints are then added together to get constraint (9), after which $\overline{y}_{K_1,2} \geq 1$ is obtained by a RUP step.

The next SAT solver call with $b_3 = b_4 = 0$ returns as core the input clause $b_3 + b_4 \geq 1$, and reformulation (lines (11)–(13)) yields $O_{ref} = 5y_{K_1,2} + y_{K_2,2} + 6$ with $LB = 6$. Now suppose the SAT solver finds the solution α_2 with $b_2 = b_3 = x = 1$ and all other variables set to 0, resulting in the objective-improving constraint (15). Since $O_{orig}(\alpha_2) = 6 = LB$, the solver terminates and reports α_2 to be optimal. To certify that this is correct, another objective reformulation constraint (16) is derived, after which the contradictory constraint (17) is obtained by adding (15) and (16). This proves that solutions with cost less than 6 do not exist.

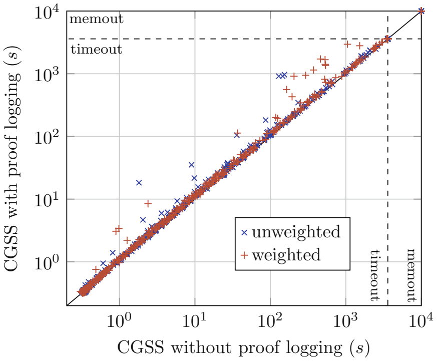


Fig. 1. Running time of CGSS with and without proof logging.

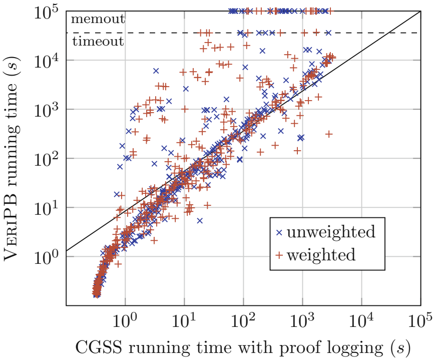


Fig. 2. CGSS running time compared to time required for proof checking.

5 Experimental Evaluation

To evaluate the proof logging techniques developed in this paper, we have implemented them in the state-of-the-art MaxSAT solver CGSS [22,47], which uses the OLL algorithm and structure-sharing totalizers. We employed VeriPB [76], extended to parse MaxSAT instances in the standard WCNF format, to verify the certificates of correctness emitted by the certifying solver.

Our experiments were conducted on machines with an 11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60 GHz CPU and 16 GB of memory. Each benchmark ran exclusively on a single machine with a memory limit of 14 GB and a time limit of 3 600 s for solving with CGSS and 36 000 s for checking the certificates with VeriPB. As benchmarks we used all 594 weighted and 607 unweighted instances from the complete track of the MaxSAT Evaluation 2022 [61], where an instance (F, O) is *unweighted* if all coefficients $\mathit{coeff}(O, \ell)$ are equal. The data from our experiments can be found in [12].

Overhead of Proof Logging. To evaluate the overhead in solver running time, we compared the standard CGSS solver [23] without proof logging (but with the bug fixes discussed below) to CGSS with proof logging as described in this paper. With proof logging 803 instances are solved within the resource limits, which is 3 instances less than without proof logging (see Fig. 1). Adding proof logging slowed down CGSS by about 8.8% in the median over all solved instances. For 95% of the instances CGSS with proof logging was at most 36.2% slower. Thus, the proof logging overhead seems perfectly manageable and should present no serious obstacles to using proof logging in core-guided MaxSAT solvers.

Overhead of Proof Checking. To assess the efficiency of proof checking, we compared the running time of CGSS with proof logging to the time taken by VeriPB for checking the generated proofs. The instances that were not solved

Table 2. Illustration of discovered bug (where $y_{i,k}$ should be read as $y_{K_i,k}$).

#iter	Literals considered ($w_{strat} = 2$)	Core $K_{\#\text{iter}}$ extracted
1	$\{b_i, e_i \mid i = 1 \dots 5\}$	$K_1 = \sum_{i=1}^{5} b_i \geq 1$
2	$\{e_i \mid i = 1 \dots 5\} \cup \{y_{1,2}\}$	$K_2 = y_{1,2} + e_2 + e_4 \geq 1$
3	$\{e_i \mid i = 1 \dots 3, 5\} \cup \{y_{1,2}, y_{1,3}\} \cup \{y_{2,2}\}$	$K_3 = y_{1,3} + e_1 + e_2 + e_5 \geq 1$
4	$\{e_i \mid i = 1 \dots 3\} \cup \{y_{1,2}, y_{1,4}\} \cup \{y_{2,2}, y_{3,2}\}$	$K_4 = y_{1,2} + e_1 + e_2 \geq 1$
5	$\{e_i \mid i = 1 \dots 3\} \cup \{y_{1,4}\} \cup \{y_{2,2}, y_{3,2}, y_{4,2}\}$	$K_5 = e_1 + e_2 + e_3 + y_{1,4} + y_{2,2} \geq 1$
6	$\{e_3\} \cup \{y_{1,5}\} \cup \{y_{2,3}\} \cup \{y_{3,2}, y_{4,2}, y_{5,2}\}$	Result is SAT

#iter	O_{ref} (after reformulation of $K_{\#\text{iter}}$)
0	$10\left(\sum_{i=1}^{5} b_i\right) + 11e_1 + 14e_2 + 11e_3 + 3e_4 + 2e_5 + o_1 + o_2$
1	$11e_1 + 14e_2 + 11e_3 + 3e_4 + 2e_5 + 10y_{1,2} + o_1 + o_2 + 10$
2	$11e_1 + 11e_2 + 11e_3 + 2e_5 + 7y_{1,2} + 3y_{1,3} + 3y_{2,2} + o_1 + o_2 + 13$
3	$9e_1 + 9e_2 + 11e_3 + 7y_{1,2} + y_{1,3} + 2y_{1,4} + 3y_{2,2} + 2y_{3,2} + o_1 + o_2 + 15$
4	$2e_1 + 2e_2 + 11e_3 + \mathbf{8y_{1,3}} + 2y_{1,4} + 3y_{2,2} + 2y_{3,2} + 7y_{4,2} + o_1 + o_2 + 22$
5	$9e_3 + \mathbf{8y_{1,3}} + 2y_{1,5} + y_{2,2} + 2y_{2,3} + 2y_{3,2} + 7y_{4,2} + 2y_{5,2} + o_1 + o_2 + 24$

by CGSS within the resource limits were filtered out, since the running time for checking an incomplete proof is inconclusive.

VeriPB successfully checked the proofs for 747 out of the 803 instances solved by CGSS (see Fig. 2); 42 instances failed due to the memory limit and 14 instances failed due to the time limit. Checking the proof took about 3 times the solving time in the median for successfully checked instances. About 87% of the successfully checked instances were checked within 10 times the solving time.

Proof checking time compared to solver running time varies widely, but our experiments indicate that the performance of VeriPB is sufficient in most cases, and verification time scales linearly with the size of the proof for a majority of the instances. However, there is room to improve VeriPB, where focus so far has been on proof logging strength rather than performance. For the instances where checking is 100 times slower than solving, the main bottleneck is the proof generated by the SAT solver, which could be addressed by standard techniques for checking *DRAT* proofs, and checking logged solutions (when objective improving constraints (3a) are added) could also be implemented more efficiently.

Bugs Discovered by Proof Logging. Our work on implementing proof logging in CGSS led to the discovery of two bugs, which were also present in the solver RC2 on which CGSS is based, but have now been fixed in CGSS in commit `5526d04` and in RC2 in commit `d0447c3`. The bugs are due to a slightly different implementation of OLL compared to the description in Sect. 3.

First, when a counting variable $y_{K_{old},i}$ for a core K_{old} appears for the first time in a later core K_{new}, the next counting variable $y_{K_{old},i+1}$ is added to the reformulated objective with coefficient $w\left(K_{new}, O_{new}\right)$ rather than $w\left(K_{old}, O_{old}\right)$. The coefficient of $y_{K_{old},i+1}$ is then further increased when $y_{K_{old},i}$ is found in future cores. Second, rather than computing the upper bound *UB* from an actual solution, CGSS uses a weaker estimate UB_{est} obtained by summing the current lower bound and the coefficients of all literals b where $\mathit{coeff}(O_{ref}, b) < w_{strat}$ (meaning that these literals were not set to 0 in the SAT solver call, and so could potentially be true in the solution).

The bugs we detected could lead to the solver producing an overly optimistic estimate $UB_{est} < UB$. The first way this can happen is when the contributions of counting variables $y_{K,k}$ in the reformulated objective are underestimated due to too small coefficients. The second bug is when the coefficient of $y_{K_{old},i+1}$ is first lowered below w_{strat} and then raised above this threshold again when $y_{K_{old},i}$ is found in a core. Then CGSS fails to assume $y_{K_{old},i+1} = 0$ in future solver calls. These bugs can result in erroneous hardening as detailed in the next example.

Example 1. Given a MaxSAT instance (F, O) with $F = \left\{ \left(\bigvee_{i=1}^{5} b_i \right), (o_1 \vee o_2) \right\} \cup \{b_i \vee e_i \mid i = 1, \ldots, 5\}$ and $O = \left(\sum_{i=1}^{5} 10 \cdot b_i \right) + 11 \cdot e_1 + 14 \cdot e_2 + 11 \cdot e_3 + 3 \cdot e_4 + 2 \cdot e_5 + o_1 + o_2$, assume the stratification bound is $w_{strat} = 2$. Table 2 displays a possible CGSS run for this instance, except that for simplicity we assume one core extraction per iteration and no use of any other heuristics. The upper half of the table lists the variables set to 0 in solver calls, the extracted core, and the lower bound derived from it. The lower half of the table provides the reformulated objective. Even though the coefficient of $y_{K_1,3}$ is increased to 8 after the fourth core, this variable is not set to 0 in subsequent iterations, which allows the solver to finish the stratification level after extracting 6 cores with a solution that sets to true the variables $b_1, b_2, b_3, b_5, e_4, o_1, o_2, y_{K_2,2}$ and $y_{K_1,i}$ for $i = 1, \ldots, 4$, and all other variables to false. The cost of this solution is 45.

Now CGSS would incorrectly estimate $UB_{est} = LB + 4 = 28$, since $y_{K_1,3}$ and $y_{K_2,2}$ (abbreviated as $y_{1,3}$ and $y_{2,2}$ in the table) both have coefficient 1 in the current reformulated objective. This is lower than the cost 45 of the solution found (and even than the optimum 36), and erroneously allows hardening—which considers $y_{K_1,3}$ with the correct coefficient 8—to fix $y_{K_1,3} = 0$, even though b_1, b_2 and b_3 (and hence also $y_{K_1,3}$) are true in every minimal-cost solution.

In our computational experiments there were cases of faulty hardening, but all incorrectly fixed values happened to agree with some optimal solution and so we never observed incorrect results. Proof logging detected the problem, however, since the derivations of the buggy hardening steps failed during proof checking. Interestingly, what proof logging did *not* turn up was any examples of mistaken claims $O_{orig} \leq UB_{est} - 1$ when the cost of a found solution was estimated. The issue with mistaken estimates due to faulty stratification was instead discovered while analyzing and fixing the hardening bug. The moral of this is that even if all results are certified as correct, this does not certify that the code is free from bugs that have not yet manifested themselves. However, proof logging still guarantees that even if the solver would have undiscovered bugs, we can always trust computed results for which the accompanying proofs pass verification.

6 Concluding Remarks

In this work, we develop pseudo-Boolean proof logging techniques for core-guided MaxSAT solving and implement them in the solver CGSS [47] with support for the full range of sophisticated reasoning techniques it uses. To the best of our knowledge, this is the first time a state-of-the-art MaxSAT solver has been enhanced to output machine-verifiable proofs of correctness. We have made a thorough evaluation on benchmarks from the MaxSAT Evaluation 2022 using the VeriPB proof checker [17,42], and find that proof logging overhead is perfectly manageable and that proof verification time, while leaving room for improvement, is definitely practically feasible. Our work also showcases the benefit of proof logging as a debugging tool—erroneous proofs produced by CGSS revealed two subtle bugs in the solver that previous extensive testing had failed to uncover.

Regarding proof verification time, further investigation is needed into the rare cases where verification is much slower (say, more than a factor 10) than solving. There are reasons to believe, though, that this is not a problem of MaxSAT proof logging per se, but rather is explained by features not yet added to VeriPB, which is a tool currently undergoing very active development. So far, the proof checker has been optimized for other types of reasoning than the clausal reverse unit propagation (RUP) steps that dominate SAT proofs. Also, VeriPB lacks the ability to trim proofs during checking as in [44]. Finally, introducing a binary proof format in addition to plain-text proofs would be another way to boost performance of proof checking. But these are matters of engineering rather than research, and can be taken care of once the proof logging technology as such has been developed and has proven its worth.

The focus of this work is on core-guided MaxSAT solving, but we would like to extend our techniques to solvers using linear SAT-UNSAT (LSU) solving (such as Pacose [68]) and implicit hitting set (IHS) search (such as MaxHS [28,29]). Although there are certainly nontrivial technical challenges that will need to be overcome, we are optimistic that our work paves the way towards a unified proof logging system for the full range of modern MaxSAT solving approaches. Going beyond MaxSAT, it would also be interesting to extend VeriPB proof logging to pseudo-Boolean solvers using core-guided search [30] or IHS [73,74], and perhaps even to similar techniques in constraint programming [36] and answer set programming [5].

Acknowledgements. This work was partly carried out while some of the authors were visiting the Simons Institute for the Theory of Computing at UC Berkeley for the extended reunion of the program “Satisfiability: Theory, Practice, and Beyond” during the spring of 2023. We also benefited greatly from the Dagstuhl Seminar 22411 “Theory and Practice of SAT and Combinatorial Solving”. Additionally, we acknowledge several inspirational discussions on certifying solvers and proof logging with, among others, Ambros Gleixner, Stephan Gocht, and Ciaran McCreesh. The computational experiments were enabled by resources provided by LUNARC at Lund University.

Jeremias Berg was fully supported by the Academy of Finland under grant 342145. Bart Bogaerts and Dieter Vandesande were supported by Fonds Wetenschappelijk

Onderzoek – Vlaanderen (project G070521N) and by the EU ICT-48 2020 project TAILOR (GA 952215). Jakob Nordström was supported by the Swedish Research Council grant 2016-00782 and the Independent Research Fund Denmark grant 9040-00389B. Andy Oertel was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

1. Achterberg, T., Wunderling, R.: Mixed integer programming: analyzing 12 years of progress. In: Jünger, M., Reinelt, G. (eds.) Facets of Combinatorial Optimization, pp. 449–481. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38189-8_18
2. Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Metamorphic testing of constraint solvers. In: Hooker, J. (ed.) CP 2018. LNCS, vol. 11008, pp. 727–736. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98334-9_46
3. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C., Schweitzer, P.: An introduction to certifying algorithms. IT - Inf. Technol. Methoden Innov. Anwendungen Inform. Informationstechnik **53**(6), 287–293 (2011)
4. Alviano, M., Dodaro, C., Ricca, F.: A MaxSAT algorithm using cardinality constraints of bounded size. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015), pp. 2677–2683. AAAI Press (2015)
5. Andres, B., Kaufmann, B., Matheis, O., Schaub, T.: Unsatisfiability-based optimization in clasp. In: Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012). Leibniz International Proceedings in Informatics (LIPIcs), vol. 17, pp. 211–221 (2012)
6. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving SAT-based weighted MaxSAT solvers. In: Milano, M. (ed.) CP 2012. LNCS, pp. 86–101. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_9
7. Ansótegui, C., Gabàs, J.: WPM3: an (in)complete algorithm for weighted partial MaxSAT. Artif. Intell. **250**, 37–57 (2017)
8. Bacchus, F., Järvisalo, M., Martins, R.: Maximum satisfiabiliy. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, 2nd edn., vol. 336, pp. 929–991. IOS Press (2021)
9. Bailleux, O., Boufkhad, Y.: Efficient CNF encoding of Boolean cardinality constraints. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 108–122. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45193-8_8
10. Barth, P.: A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical report MPI-I-95-2-003, Max-Planck-Institut für Informatik (1995)
11. Bayardo Jr., R.J., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: Proceedings of the 14th National Conference on Artificial Intelligence (AAAI 1997), pp. 203–208 (1997)
12. Berg, J., Bogaerts, B., Nordström, J., Oertel, A., Vandesande, D.: Experimental repository for "Certified core-guided MaxSAT solving" (2023). https://doi.org/10.5281/zenodo.7709687
13. Berg, J., Järvisalo, M.: Weight-aware core extraction in SAT-based MaxSAT solving. In: Beck, J.C. (ed.) CP 2017. LNCS, vol. 10416, pp. 652–670. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66158-2_42

14. Biere, A.: Tracecheck (2006). http://fmv.jku.at/tracecheck/
15. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, 2nd edn., vol. 336. IOS Press (2021)
16. Bixby, R., Rothberg, E.: Progress in computational mixed integer programming–a look back from the other side of the tipping point. Ann. Oper. Res. **149**(1), 37–41 (2007)
17. Bogaerts, B., Gocht, S., McCreesh, C., Nordström, J.: Certified symmetry and dominance breaking for combinatorial optimisation. In: Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI 2022), pp. 3698–3707 (2022)
18. Bogaerts, B., McCreesh, C., Nordström, J.: Solving with provably correct results: beyond satisfiability, and towards constraint programming (2022). Tutorial at the 28th International Conference on Principles and Practice of Constraint Programming. Slides available at http://www.jakobnordstrom.se/presentations/
19. Bonet, M.L., Levy, J., Manyà, F.: Resolution for max-SAT. Artif. Intell. **171**(8–9), 606–618 (2007)
20. Buss, S.R., Nordström, J.: Proof complexity and SAT solving. In: Biere et al. [15], chap. 7, pp. 233–350 (2021)
21. Calabro, C., Impagliazzo, R., Paturi, R.: The complexity of satisfiability of small depth circuits. In: Chen, J., Fomin, F.V. (eds.) IWPEC 2009. LNCS, vol. 5917, pp. 75–85. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-11269-0_6
22. Certifying version of the CGSS core-guided MaxSAT solver with structure sharing. https://gitlab.com/MIAOresearch/software/certified-cgss
23. CGSS, a core guided Max-SAT-algorithm using structure sharing technique for enhanced cardinality constraints, built on RC2 and PySAT. https://bitbucket.org/coreo-group/cgss/
24. Cook, W., Coullard, C.R., Turán, G.: On the complexity of cutting-plane proofs. Discret. Appl. Math. **18**(1), 25–38 (1987)
25. Cook, W., Koch, T., Steffy, D.E., Wolter, K.: A hybrid branch-and-bound approach for exact rational mixed-integer programming. Math. Program. Comput. **5**(3), 305–344 (2013)
26. Cruz-Filipe, L., Heule, M.J.H., Hunt, W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 220–236. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_14
27. Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 118–135. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_7
28. Davies, J., Bacchus, F.: Exploiting the power of MIP solvers in MAXSAT. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 166–181. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39071-5_13
29. Davies, J., Bacchus, F.: Postponing optimization to speed up MAXSAT solving. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 247–262. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_21
30. Devriendt, J., Gocht, S., Demirović, E., Nordström, J., Stuckey, P.: Cutting to the core of pseudo-Boolean optimization: combining core-guided search with cutting planes reasoning. In: Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI 2021), pp. 3750–3758 (2021)
31. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. J. Satisfiab. Boolean Model. Comput. **2**(1–4), 1–26 (2006)

32. Eifler, L., Gleixner, A.: A computational status update for exact rational mixed integer programming. In: Singh, M., Williamson, D.P. (eds.) IPCO 2021. LNCS, vol. 12707, pp. 163–177. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-73879-2_12
33. Elffers, J., Gocht, S., McCreesh, C., Nordström, J.: Justifying all differences using pseudo-Boolean reasoning. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020), pp. 1486–1494 (2020)
34. Filmus, Y., Mahajan, M., Sood, G., Vinyals, M.: MaxSAT resolution and subcube sums. In: Pulina, L., Seidl, M. (eds.) SAT 2020. LNCS, vol. 12178, pp. 295–311. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51825-7_21
35. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 252–265. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_25
36. Gange, G., Berg, J., Demirović, E., Stuckey, P.J.: Core-guided and core-boosted search for CP. In: Hebrard, E., Musliu, N. (eds.) CPAIOR 2020. LNCS, vol. 12296, pp. 205–221. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58942-4_14
37. Gillard, X., Schaus, P., Deville, Y.: SolverCheck: declarative testing of constraints. In: Schiex, T., de Givry, S. (eds.) CP 2019. LNCS, vol. 11802, pp. 565–582. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30048-7_33
38. Gocht, S., Martins, R., Nordström, J., Oertel, A.: Certified CNF translations for pseudo-Boolean solving. In: Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 236, pp. 16:1–16:25 (2022)
39. Gocht, S., McBride, R., McCreesh, C., Nordström, J., Prosser, P., Trimble, J.: Certifying solvers for clique and maximum common (connected) subgraph problems. In: Simonis, H. (ed.) CP 2020. LNCS, vol. 12333, pp. 338–357. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58475-7_20
40. Gocht, S., McCreesh, C., Nordström, J.: Subgraph isomorphism meets cutting planes: solving with certified solutions. In: Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI 2020), pp. 1134–1140 (2020)
41. Gocht, S., McCreesh, C., Nordström, J.: An auditable constraint programming solver. In: Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 235, pp. 25:1–25:18 (2022)
42. Gocht, S., Nordström, J.: Certifying parity reasoning efficiently using pseudo-Boolean proofs. In: Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI 2021), pp. 3768–3777 (2021)
43. Goldberg, E., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2003), pp. 886–891 (2003)
44. Heule, M.J.H., Hunt Jr., W.A., Wetzler, N.: Trimming while checking clausal proofs. In: Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2013), pp. 181–188 (2013)
45. Heule, M.J.H., Hunt, W.A., Wetzler, N.: Verifying refutations with extended resolution. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 345–359. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_24
46. Ignatiev, A., Morgado, A., Marques-Silva, J.P.: RC2: an efficient MaxSAT solver. J. Satisfiab. Boolean Model. Comput. **11**(1), 53–64 (2019)

47. Ihalainen, H., Berg, J., Järvisalo, M.: Refined core relaxation for core-guided MaxSAT solving. In: 27th International Conference on Principles and Practice of Constraint Programming (CP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 210, pp. 28:1–28:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
48. Ihalainen, H., Berg, J., Järvisalo, M.: Clause redundancy and preprocessing in maximum satisfiability. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 75–94. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_6
49. Impagliazzo, R., Paturi, R.: On the complexity of k-SAT. J. Comput. Syst. Sci. **62**(2), 367–375 (2001). Preliminary version in CCC 1999
50. Karpinski, M., Piotrów, M.: Competitive sorter-based encoding of PB-constraints into SAT. In: Proceedings of Pragmatics of SAT. EPiC Series in Computing, vol. 59, pp. 65–78. EasyChair (2018)
51. Karpinski, M., Piotrów, M.: Encoding cardinality constraints using multiway merge selection networks. Constraints **24**(3–4), 234–251 (2019)
52. Kraiczy, S., McCreesh, C.: Solving graph homomorphism and subgraph isomorphism problems faster through clique neighbourhood constraints. In: Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI 2021), pp. 1396–1402 (2021)
53. Larrosa, J., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: A framework for certified Boolean branch-and-bound optimization. J. Autom. Reason. **46**(1), 81–102 (2011)
54. Le Berre, D., Parrain, A.: The Sat4j library, release 2.2. J. Satisfiab. Boolean Model. Comput. **7**, 59–64 (2010)
55. Leivo, M., Berg, J., Järvisalo, M.: Preprocessing in incomplete MaxSAT solving. In: Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020). Frontiers in Artificial Intelligence and Applications, vol. 325, pp. 347–354. IOS Press (2020)
56. Li, C.M., Manyà, F.: MaxSAT, hard and soft constraints. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 903–927. IOS Press (2021)
57. Li, C., Xu, Z., Coll, J., Manyà, F., Habet, D., He, K.: Boosting branch-and-bound MaxSAT solvers with clause learning. AI Commun. **35**(2), 131–151 (2022)
58. Marques-Silva, J., Argelich, J., Graça, A., Lynce, I.: Boolean lexicographic optimization: algorithms & applications. Ann. Math. Artif. Intell. **62**(3–4), 317–343 (2011)
59. Marques-Silva, J.P., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999). Preliminary version in ICCAD 1996
60. Martins, R., Joshi, S., Manquinho, V., Lynce, I.: Incremental cardinality constraints for MaxSAT. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 531–548. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_39
61. MaxSAT evaluation 2022 (2022). https://maxsat-evaluations.github.io/2022
62. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Comput. Sci. Rev. **5**(2), 119–161 (2011)
63. Morgado, A., Dodaro, C., Marques-Silva, J.: Core-guided MaxSAT with soft cardinality constraints. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 564–573. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_41

64. Morgado, A., Ignatiev, A., Bonet, M.L., Marques-Silva, J., Buss, S.: DRMaxSAT with MaxHS: first contact. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 239–249. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_17
65. Morgado, A., Marques-Silva, J.: On validating Boolean optimizers. In: Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence, (ICTAI 2011), pp. 924–926 (2011)
66. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC 2001), pp. 530–535 (2001)
67. Narodytska, N., Bacchus, F.: Maximum satisfiability using core-guided MaxSAT resolution. In: Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI 2014), pp. 2717–2723. AAAI Press (2014)
68. Paxian, T., Reimer, S., Becker, B.: Dynamic polynomial watchdog encoding for solving weighted MaxSAT. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 37–53. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_3
69. Py, M., Cherif, M.S., Habet, D.: Towards bridging the gap between SAT and Max-SAT refutations. In: Proceedings of the 32nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2020), pp. 137–144 (2020)
70. Py, M., Cherif, M.S., Habet, D.: A proof builder for Max-SAT. In: Li, C.-M., Manyà, F. (eds.) SAT 2021. LNCS, vol. 12831, pp. 488–498. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-80223-3_33
71. Py, M., Cherif, M.S., Habet, D.: Proofs and certificates for Max-SAT. J. Artif. Intell. Res. **75**, 1373–1400 (2022)
72. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol. 2. Elsevier, Amsterdam (2006)
73. Smirnov, P., Berg, J., Järvisalo, M.: Improvements to the implicit hitting set approach to pseudo-Boolean optimization. In: Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 236, pp. 13:1–13:18 (2022)
74. Smirnov, P., Berg, J., Järvisalo, M.: Pseudo-Boolean optimization by implicit hitting sets. In: Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 210, pp. 51:1–51:20 (2021)
75. Vandesande, D., De Wulf, W., Bogaerts, B.: QMaxSATpb: a certified MaxSAT solver. In: Gottlob, G., Inclezan, D., Maratea, M. (eds.) LPNMR 2022. LNCS, vol. 13416, pp. 429–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-15707-3_33
76. VeriPB: Verifier for pseudo-Boolean proofs. https://gitlab.com/MIAOresearch/software/VeriPB

Superposition with Delayed Unification

Ahmed Bhayat[1](✉), Johannes Schoisswohl[2], and Michael Rawson[2]

[1] University of Manchester, Manchester, UK
ahmed.bhayat@manchester.ac.uk
[2] TU Wien, Vienna, Austria
{johannes.schoisswohl,michael.rawson}@tuwien.ac.at

Abstract. Classically, in saturation-based proof systems, unification has been considered atomic. However, it is also possible to move unification to the calculus level, turning the steps of the unification algorithm into inferences. For calculi that rely on unification procedures returning large or even infinite sets of unifiers, integrating unification into the calculus is an attractive method of dovetailing unification and inference. This applies, for example, to AC-superposition and higher-order superposition. We show that first-order superposition remains complete when moving unification rules to the calculus level. We discuss some of the benefits this has even for standard first-order superposition and provide an experimental evaluation.

1 Introduction

Unification is a key feature in many proof calculi, particularly those based on the saturation framework. It acts as a filter, reducing the number of inferences that need to be carried out by instantiating terms only to the degree necessary. However, many unification algorithms have large time complexities and produce large, or even infinite, sets of unifiers. This is the case, for example, for AC-unification, which can produce a doubly exponential number of unifiers [10], and higher-order unification, which can produce an infinite set of unifiers [20]. This motivates the study of how unification rules can be integrated into proof calculi to allow them to dovetail with standard calculus rules. One way to achieve this is to use the concept of unification with abstraction [13,17]. The general idea is that during the unification process, instead of solving all unification pairs, certain pairs are retained and added to the conclusion of an inference as negative *constraint* literals. Calculus-level unification inferences then work on such literals to solve these constraints and remove the literals in the case they are unifiable. Note how this differs from constrained resolution-style calculi such as [4,15] where the constraints are completely separate from the rest of the clause and are not subject to inferences.

To demonstrate the idea of dedicated unification inferences in combination with unification with abstraction, we provide the following example.

$$C_1 = f(g(a,x)) \not\approx t \qquad C_2 = f(g(a,b)) \approx t$$

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 23–40, 2023.
https://doi.org/10.1007/978-3-031-38499-8_2

A standard superposition calculus would proceed by unifying $f(g(a, b))$ and $f(g(a, x)$ with the unifier $\sigma = \{x \to b\}$ and then rewriting C_1 with C_2 to derive $t\sigma \not\approx t\sigma$. Equality resolution on $t\sigma \not\approx t\sigma$ would then derive $\bot$. It is also possible to proceed by rewriting C_1 with C_2 *without* computing σ and instead add the constraint literal $g(a, x) \not\approx g(a, b)$ to the conclusion to derive $t \not\approx t \vee g(a, x) \not\approx g(a, b)$. A dedicated unification inference could then decompose the constraint literal resulting in $t \not\approx t \vee a \not\approx a \vee b \not\approx x$. Further unification inferences could bind x to b, and remove the trivial pairs $a \not\approx a$ and $t \not\approx t$ to derive $\bot$.

In this paper, we investigate moving unification to the calculus level for standard first-order superposition. Whilst this may seem like a regressive step, as we lose much of unification's power to act as a filter on inferences and hence produce many more clauses, we think the investigation is valuable for two reasons.

Firstly, by showing how syntactic first-order unification can be lifted to the calculus level, we provide a roadmap for how more complex unification problems can be lifted to the calculus level. This may prove particularly useful in the higher-order case, where abstraction may expose terms to standard calculus rules that were unavailable before. Moreover, we note that in our calculus we do not turn the entire unification problem into a constraint, but rather a subproblem. Whilst this may be merely an interesting detail for first-order unification, for more complex unification problems, such a method could be used to eagerly solve simple unification subproblems whilst delaying complex subproblems by adding them as constraints.

Secondly, one of the most expensive operations in first-order theorem provers is the maintenance of indices. Indices are crucial to the performance of modern solvers, as they facilitate the efficient retrieval of terms unifiable or matchable with a query term. However, solvers typically spend a large amount of time inserting and removing terms from indices as well as unifying against terms in the indices. This is particularly the case in the presence of the AVATAR architecture [24] wherein a change in the model can trigger the insertion and removal of thousands of terms from various indices. By moving unification to the calculus level, we can replace complex indices with simple hash maps, since to trigger an inference we merely need to check for top symbol equality and not unifiability. Insertion and deletion become $O(1)$ time operations. However, for first-order logic, we do not expect the time gained to offset the downsides of extra inferences carried out and extra clauses created. Our experimental results back up this hypothesis (see Sect. 7). Our main contributions are:

- Designing a modified superposition calculus that moves unification to the calculus level (Sect. 3).
- Proving the calculus to be statically and dynamically refutationally complete (Sect. 5).
- Providing a thorough empirical evaluation of the calculus (Sect. 7).

2 Preliminaries

Syntax. We consider standard monomorphic first-order logic with equality. We assume a signature consisting of a finite set of (monomorphically) typed function symbols and a single predicate, equality, denoted by $\approx$. A non-equality atom A can be expressed using equality as $A \approx \top$ where $\top$ is a special function symbol [18]. Terms are formed in the normal way from variables and function symbols. We commonly use s, t or u or their primed variants to refer to terms. We write $s : \tau$ to show that term s has type τ. A term is ground if it contains no variables. We use the notation $\overline{s}_n$ to refer to a tuple or list of terms of length n. More generally, we use the over bar notation to refer to tuples and lists of various objects. Where the length of the tuple or list is not relevant, we drop the subscript. By s_i we denote the ith element of the tuple $\overline{s}_n$. Literals are positive or negative equalities written as $s \approx t$ and $s \not\approx t$ respectively. We use $s \dot{\approx} t$ to refer to either a positive or a negative equality. Clauses are multisets of literals. A clause that contains no literals is known as the empty clause and denoted $\bot$.

A substitution is a mapping from variables to terms. We assume, w.l.o.g., that all substitutions are idempotent. We commonly denote substitutions using σ and θ and denote the application of a substitution σ to a term s by $s\sigma$. A substitution θ is grounding for a term s, if $s\theta$ is ground. The definition of grounding substitution can be extended to literals and clauses in the obvious manner. A substitution σ is a unifier of terms s and t if $s\sigma = t\sigma$. A unifier σ is more general than a unifier σ' if there exists a substitution ρ such that $\sigma\rho = \sigma'$. With respect to syntactic first-order unification, if two terms are unifiable then they have a single most general unifier up to variable naming [1].

A transitive irreflexive relation over terms is known as an ordering. The superposition calculus we present below is, as usual, parameterised by a simplification ordering on ground terms. An ordering $\succ$ is a simplification ordering, if it possesses the following properties. It is total on ground terms. It is compatible with contexts, meaning that if $s \succ t$, then $u[s] \succ u[t]$. It is well-founded. Note that every simplification ordering has the subterm property. Namely, that if t is a proper subterm of s, then $s \succ t$. For non-ground terms, the only property that is required of the ordering is that it is stable under substitution. That is, if $s \succ t$ then for all substitutions σ, $s\sigma \succ t\sigma$. We extend the ordering $\succ$ to literals in the standard fashion via its multiset extension. A positive literal $s \approx s'$ is treated as the multiset $\{s, s'\}$, whilst a negative literal $s \not\approx s'$ is treated as the multiset $\{s, s, s', s'\}$. The ordering is extended to clauses by its two-fold multiset extension. We use $\succ$ to denote the ordering on terms and its multiset extensions to literals and clauses.

Semantics. An interpretation is a pair $(\mathcal{U}, \mathcal{I})$, where $\mathcal{U}$ is a set of typed universes and $\mathcal{I}$ is an interpretation function, such that for each function symbol $f : \tau_1 \times \cdots \times \tau_n \rightarrow \tau$ in the signature, $\mathcal{I}(f)$ is a concrete function of type $\mathcal{U}_{\tau_1} \times \cdots \times \mathcal{U}_{\tau_n} \rightarrow \mathcal{U}_\tau$. A valuation ξ is a function that maps each variable $x : \tau$ to a member of $\mathcal{U}_\tau$. For a given interpretation $\mathcal{M}$ and valuation ξ, we uses $[\![t]\!]^\xi_{\mathcal{M}}$ to represent the denotation of t in $\mathcal{M}$ given ξ. A positive literal $s \approx t$ is true in an interpretation $\mathcal{M}$ for valuation ξ if $[\![s]\!]^\xi_{\mathcal{M}} = [\![t]\!]^\xi_{\mathcal{M}}$ and false otherwise. A negative literal $s \not\approx t$ is true in an interpretation $\mathcal{M}$ for valuation ξ if $s \approx t$ is false. A clause C holds in an interpretation $\mathcal{M}$ for valuation ξ if one of its literals is true in $\mathcal{M}$ for ξ. An interpretation $\mathcal{M}$ *models* a clause C if C holds in $\mathcal{M}$ for every valuation. An interpretation models a clause set, if it models every clause in the set. A set of clauses M entails a set of clauses N, denoted $M \models N$, if every model of M is also a model of N.

3 Calculus

Intuitively, what we are aiming for with our calculus, is that whenever standard superposition applies a substitution σ to a conclusion with the side condition "σ is a unifier of terms t_1 and t_2", our calculus adds a constraint $t_1 \not\approx t_2$ to the conclusion. The calculus then has further inference rules that mimic the steps of a first-order unification algorithm and work on negative literals. Our presentation below does not quite follow this intuition. Instead, if the unification problem is trivial we solve it immediately. If it is non-trivial, we carry out a single step of unification and add the resulting sub-problems as constraints. Our reasons for doing this are two-fold.

1. Adding the entire unification problem $t_1 \not\approx t_2$ as a constraint can lead to a constraint literal that is larger, with respect to $\succ$, than any literal occurring in the premises. This causes difficulties in the completeness proof.
2. More pertinently, keeping in mind our planned applications to more complex logics, we wish to show that delayed unification remains complete even when only selected sub-problems of the original unification problem are added as constraints. In the context of higher-order logic, for example, this could allow for the eager solving of simple unification sub-problems whilst only the most difficult are added as constraints. See Sect. 6 for further details.

Wherever we present a clause as a subclause C' and a literal l (e.g. $C' \vee l$), we denote the entire clause by the same name as the subclause without the dash (e.g. we refer to the clause $C' \vee l$ by C). As in the classical superposition calculus, our calculus is parameterised by a *selection function* that is used to restrict the number of applicable inferences in order to avoid the search space growing unnecessarily. A selection function *sel* is a function that maps a clause to a subset of its negative literals. We say that literal l is *σ-eligible* in a clause $C' \vee l$ if it is selected in C ($l \in sel(C)$), or there are no selected literals and $l\sigma$ is maximal in $C\sigma$. Strict σ-eligibility is defined in a like fashion, with maximality replaced by strict maximality. Where σ is empty, we sometimes speak of eligibility instead of σ-eligibility. In what follows, $\mathcal{CS}$ is a multiset of literals that we refer to as *constraints.*

$$\frac{D' \vee f(\bar{t}_n) \approx t' \qquad C' \vee s[f(\bar{s}_n)] \,\dot{\approx}\, s'}{C' \vee D' \vee s[t'] \,\dot{\approx}\, s' \vee \mathcal{CS}}\ \textsc{Sup}$$

$$\frac{D' \vee x \approx t' \qquad C' \vee s[f(\bar{s}_n)] \,\dot{\approx}\, s'}{(C' \vee D' \vee s[t'] \,\dot{\approx}\, s')\sigma}\ \text{V}\textsc{Sup}$$

where $\sigma = \{x \rightarrow f(\overline{s}_n)\}$, and $\mathcal{CS} = t_1 \not\approx s_1 \vee \ldots \vee t_n \not\approx s_n$. Both rules share the following side conditions. Let t stand for either $f(\overline{t}_n)$ or x. For Sup, the substitution σ mentioned in the side conditions is of course empty.

- $t \approx t'$ is strictly σ-eligible.
- $s[f(\overline{s}_n)] \dot{\approx} s'$ is strictly σ-eligible if positive and σ-eligible if negative.
- $t\sigma \not\preceq t'\sigma$ and $s[f(\overline{s}_n)]\sigma \not\preceq s'\sigma$.
- $C\sigma \not\preceq D\sigma$

$$\frac{C' \vee f(\overline{t}_n) \approx v' \vee f(\overline{s}_n) \approx v}{C' \vee v \not\approx v' \vee f(\overline{s}_n) \approx v \vee \mathcal{CS}}\ \text{EqFact} \qquad \frac{C' \vee u' \approx v' \vee u \approx v}{(C' \vee v \not\approx v' \vee u \approx v)\sigma}\ \text{VEqFact}$$

for EqFact, $\mathcal{CS} = t_1 \not\approx s_1 \vee \ldots \vee t_n \not\approx s_n$. For VEqFact, either u or u' must be a variable and σ is the most general unifier of u and u'. The side conditions for EqFact are:

- $f(\overline{s}_n) \approx v$ be eligible in C.
- $f(\overline{s}_n) \not\preceq v$ and $f(\overline{t}_n) \not\preceq v'$.

The side conditions for VEqFact are:

- $u \approx v$ be σ-eligible in C.
- $u\sigma \not\preceq v\sigma$ and $u'\sigma \not\preceq v'\sigma$.

The calculus also contains the following resolution/unification inferences. We refer to these as unification inferences, because each inference represents carrying out a single step of the well-known Robinson unification algorithm [11].

$$\frac{C' \vee f(\overline{s}_n) \not\approx f(\overline{t}_n)}{C' \vee \mathcal{CS}}\ \text{Decompose}$$

$$\frac{C' \vee x \not\approx t}{C'\sigma}\ \text{Bind} \qquad\qquad \frac{C' \vee s \not\approx s}{C'}\ \text{ReflDel}$$

where for Bind, $\sigma = \{x \rightarrow t\}$ and x does not occur in t. For Decompose, $f(\overline{s}_n) \neq f(\overline{t}_n)$ and $\mathcal{CS} = t_1 \not\approx s_1 \vee \ldots \vee t_n \not\approx s_n$. All three inferences require that the final literal be σ-eligible in $C\sigma$ (for Decompose and ReflDel, σ is empty). We provide some examples to show how the calculus works.

Example 1. Consider the unsatisfiable clause set:

$$C_1 = f(x, g(x)) \not\approx t \qquad C_2 = f(g(b), y) \approx t$$

A Sup inference between C_1 and C_2 results in clause $C_3 = t \not\approx t \vee x \not\approx g(b) \vee g(x) \not\approx y$. A ReflDel inference on C_3 results in the clause $C_4 = x \not\approx g(b) \vee g(x) \not\approx y$. An application of Bind on C_4 with $\sigma = \{x \rightarrow g(b)\}$ results in $C_5 = g(g(b)) \not\approx y$. Another application of Bind, then leads to $\perp$.

Example 2. Consider the unsatisfiable clause set:

$$C_1 = x \approx c \qquad C_2 = f(a,c) \not\approx t \qquad C_3 = f(c,c) \approx t$$

A VSup inference between C_1 and C_2 results in clause $C_4 = f(c,c) \not\approx t$. A Sup inference between C_3 and C_4 results in the clause $C_5 = t \not\approx t \vee c \not\approx c \vee c \not\approx c$. A triple application of ReflDel starting from C_5 derives $\bot$.

Note 1. We abuse terminology and use *inference* and *inference rule* to refer both to schemas such as shown above, as well as concrete instances of such schemas. Given an inference ι, we refer to the tuple of its premises by $\mathit{prems}(\iota)$, to its maximal premise by $\mathit{mprem}(\iota)$, and to its conclusion by $\mathit{concl}(\iota)$.

4 Redundancy Criterion

We utilise Waldmann et al.'s framework [25] for proving the completeness of our calculus. Hence, our redundancy criterion is based on their intersected lifted criterion. In instantiating the framework, we roughly follow Bentkamp et al. [6]. Let the calculus defined above be referred to as *Inf*. We introduce a ground inference system *GInf* that coincides with standard superposition [3]. That is, it contains the well known three inferences, Sup, EqFact and EqRes. We refer to these inferences by GSup, GEqFact and GEqRes to indicate that they are only applied to ground clauses. Following the notation of the framework, we write $\mathit{Inf}(N)$ ($\mathit{GInf}(N)$) to denote the set of all *Inf* (*GInf*) inferences with premises in a clause set N. We introduce a grounding function $\mathcal{G}$ that maps terms, literals and clauses to the sets of their ground instances. For example, given a clause C, $\mathcal{G}(C)$ is the set $\{C\theta \mid \theta \text{ is a grounding substitution}\}$. We extend the function $\mathcal{G}$ to clause sets by letting $\mathcal{G}(N) = \bigcup_{C \in N} \mathcal{G}(C)$ where N is a set of clauses.

A ground clause C is redundant with respect to a set of ground clauses N if there are clauses $C_1, \ldots, C_n \in N$ such that for $1 \leq i \leq n$, $C_i \prec C$ and $C_1, \ldots, C_n \models C$. The set of all ground clauses redundant with respect to a set of ground clauses N is denoted $\mathit{GRed}_{Cl}(N)$.

A clause C is redundant with respect to a set of clauses N, if for every $D \in \mathcal{G}(C)$, D is redundant with respect to $\mathcal{G}(N)$ or there is a clause $C' \in N$ such that $D \in \mathcal{G}(C')$ and $C \sqsupset C'$ where $\sqsupset$ is the strict subsumption relation. That is $C \sqsupset C'$ if C is subsumed by C', but C' is not subsumed by C. The set of all clauses redundant with respect a set of clauses N is denoted $\mathit{Red}_{Cl}(N)$.

In order to define redundant inferences, we have to pay careful attention to selection functions. For non-ground clauses, we fix a selection function sel. We then let $\mathcal{G}(\mathit{sel})$ be a set of selection functions on ground clauses with the following property. For each $\mathit{gsel} \in \mathcal{G}(\mathit{sel})$, for every ground clause C, there exists a clause D such that $C \in \mathcal{G}(D)$ and the literals selected in C by gsel correspond to those selected in D by sel. We write $\mathit{GInf}^{\mathit{gsel}}$ to show that the ground inference system *GInf* is parameterised by the selection function gsel. Let ι be an inference in *Inf*. We extend the grounding function $\mathcal{G}$ to a family of grounding functions $\mathcal{G}^{\mathit{gsel}}$

for each $gsel \in \mathcal{G}(sel)$. Each function $\mathcal{G}^{gsel}$ maps terms, literals and clauses as above, and maps members of *Inf* to subsets of GInf^{gsel} as follows.[1]

Definition 1 (Ground Instance of an Inference). *Let ι be of the form $C_1, \ldots, C_n \vdash E \vee \mathcal{CS}$. An inference $\iota_g \in \mathit{GInf}^{gsel}$ is in $\mathcal{G}^{gsel}(\iota)$ if it is of the form $C_1\theta, \ldots, C_n\theta \vdash E\theta$ for some grounding substitution θ. In this case, we say that ι_g is the θ-ground instance of ι. Note that we ignore the constraints in the definition of ground instances.*

A ground inference $C_1, \ldots, C_n, C \vdash E$ with maximal premise C is redundant with respect to a clause set N if for $1 \le i \le n$, $C_i \in \mathit{GRed}_{Cl}(N)$ or $C \in \mathit{GRed}_{Cl}(N)$ or there exist clauses $D_1, \ldots D_m \in N$ such that for $1 \le i \le m$, $D_i \prec C$ and $D_1, \ldots, D_m \models E$. The set of all ground inferences redundant with respect to a set N is denoted $\mathit{GRed}_I^{gsel}(N)$.

An inference ι is redundant with respect to a clause set N if for every $gsel \in \mathcal{G}(sel)$ and for every $\iota' \in \mathcal{G}^{gsel}(\iota)$, $\iota' \in \mathit{GRed}_I^{gsel}(\mathcal{G}(N))$. In words, every ground instance of the inference is redundant with respect to $\mathcal{G}(N)$. We denote the set of all redundant inferences with respect to a set N as $\mathit{Red}_I(N)$.

A clause set N is saturated up to redundancy by an inference system *Inf* if every member of $\mathit{Inf}(N)$ is redundant with respect to N.

Note 2. Given the definition of clause redundancy above, the ReflDel inference can be utilised as a *simplification* inference. That is, the conclusion of the inference renders the premise redundant.

5 Refutational Completeness

To prove refutational completeness we utilise the above mentioned framework of Waldmann et al. [25]. In particular, we use Theorem 14 from the paper to lift completeness from the ground level to the non-ground level. We bring Theorem 14 here for clarity and to keep the paper self contained. We then present it in our notation. Let $\mathit{GRed} = (\mathit{GRed}_I^{gsel}, \mathit{GRed}_{Cl})$ and $\mathit{Red} = (\mathit{Red}_I, \mathit{Red}_{Cl})$.

Theorem 14 (from Waldmann et al. [25]). *If $(\mathit{GInf}^q, \mathit{Red}^q)$ is statically refutationally complete w.r.t. $\models^q$ for every $q \in Q$ and if for every $N \subseteq \mathbf{F}$ that is saturated w.r.t. FInf and $\mathit{Red}^{\cap\mathcal{G}}$ there exists a q such that $\mathit{GInf}^q(\mathcal{G}^q(N)) \subseteq \mathcal{G}^q(\mathit{FInf}(N)) \cup \mathit{Red}_I^q(\mathcal{G}^q(N))$, then $(\mathit{FInf}, \mathit{Red}^{\cap\mathcal{G}})$ is statically refutationally complete w.r.t. $\models^{\cap}_{\mathcal{G}}$.*

Theorem 14 (from Waldmann et al. in our Notation). *If $(\mathit{GInf}^{gsel}, \mathit{GRed})$ is statically refutationally complete w.r.t. $\models$ for every $gsel \in \mathcal{G}(sel)$ and if for every clause set N that is saturated w.r.t. Inf and Red there exists a gsel such that $\mathit{GInf}^{gsel}(\mathcal{G}^{gsel}(N)) \subseteq \mathcal{G}^{gsel}(\mathit{Inf}(N)) \cup \mathit{Red}_I(\mathcal{G}^{gsel}(N))$, then $(\mathit{Inf}, \mathit{Red})$ is statically refutationally complete w.r.t. $\models_{\mathcal{G}}$.*

[1] When a grounding function $\mathcal{G}^{gsel}$ acts on a clause, literal or term, we commonly drop the *gsel* superscript as the selection function plays no role in the grounding of these.

Thus, in our context, the set Q is $\mathcal{G}(sel)$, the ground inference system $GInf^q$ maps to $GInf^{gsel}$, the ground redundancy criterion Red^q is $(GRed_I^{gsel}, GRed_{Cl})$ and the ground entailment relation $\models^q$ maps to standard entailment on first-order clauses. Moreover, the non-ground inference system *FInf* maps to *Inf* and the redundancy criterion $Red^{\cap\mathcal{G}}$ maps to (Red_I, Red_{Cl}). Note, that this final mapping is not exact, as the criterion $Red^{\cap\mathcal{G}}$ does not allow for a tiebreaker ordering, such as the strict subsumption relation, to be utilised in the definition of non-ground redundancy. However, this mismatch can easily be repaired since Theorem 16 of the framework paper extends the result of Theorem 14 to the case where tiebreaker orderings are used.

As our ground inference systems $GInf^{gsel}$ are ground superposition systems, static refutational completeness with respect to standard entailment and standard redundancy is a famous result. See for example [2]. What remains for us to prove in order to apply Theorem 14 and show the static refutational completeness of *Inf*, is:

1. For every $gsel \in \mathcal{G}(sel)$, the grounding function $\mathcal{G}^{gsel}$ is a grounding function in the sense of the framework.
2. For every clause set N saturated up to redundancy by *Inf*, there exists a $gsel \in \mathcal{G}(sel)$ such that $GInf^{gsel}\ (\mathcal{G}(N)) \subseteq \mathcal{G}^{gsel}(Inf(N)) \cup GRed_I^{gsel}(\mathcal{G}(N))$. In words, there exists a ground selection function such that every ground inference with that selection function and premises in $\mathcal{G}(N)$ is either the instance of a non-ground inferences with premises in N or is redundant with respect to $\mathcal{G}(N)$.

Lemma 1. *For every gsel* $\in \mathcal{G}(sel)$*, the grounding function* $\mathcal{G}^{gsel}$ *is a grounding function in the sense of the framework.*

Proof. We need show that properties (G1) – (G3) defined by Waldmann et al. hold for grounding functions. These properties are:

(G1) for every $\bot \in \mathbf{F}_\bot$, $\emptyset \neq \mathcal{G}(\bot) \subseteq \mathbf{G}_\bot$;
(G2) for every $C \in \mathbf{F}$, if $\bot \in \mathcal{G}(C)$ and $\bot \in (G)_\bot$ then $C \in \mathbf{F}_\bot$;
(G3) for every $\iota \in FInf$, if $\mathcal{G}(\iota) \neq undef$, then $\mathcal{G}(\iota) \subseteq Red_I(\mathcal{G}(concl(\iota)))$.

As properties (G1) and (G2) relate to the grounding of terms and clauses, and our grounding of these is fully standard we skip these. We prove (G3), which in our terminology is: for every $\iota \in Inf$, $\mathcal{G}^{gsel}(\iota) \subseteq GRed_I^{gsel}(\mathcal{G}(concl(\iota)))$. This can be achieved by showing that for every $\iota' \in \mathcal{G}^{gsel}(\iota)$, there exist clauses $\overline{C} \in \mathcal{G}(concl(\iota))$ such that $\overline{C} \models concl(\iota')$ and for each $C_i \in \overline{C}$, $C_i \prec mprem(\iota')$. In what follows, let θ be the substitution by which ι' is a grounding of ι.

If $\mathcal{CS}$ is the empty set in $concl(\iota)$, then $concl(\iota)\theta = concl(\iota')$ and hence $concl(\iota)\theta \models concl(\iota')$. Moreover, $concl(\iota)\theta \in \mathcal{G}(concl(\iota))$ and thus $concl(\iota)\theta \prec mprem(\iota')$.

On the other hand, if $\mathcal{CS}$ is not empty, let $u = f(\overline{t}_n)$ and $u' = f(\overline{s}_n)$ be the two terms within $prems(\iota)$ from which the constraints are created. By the existence of ι', we have that $u\theta = u'\theta$, and hence that $t_i\theta = s_i\theta$ for $1 \leq i \leq n$. Hence, every literal in $\mathcal{CS}\theta$ has the form $t \not\approx t$ and is trivially false in every interpretation. Thus, we still have $concl(\iota)\theta \models concl(\iota')$. Moreover, by the subterm property of the ordering $\succ$ we have that $t_i\theta \not\approx s_i\theta$ is smaller than the maximal/selected literal of $mprem(\iota')$ for $1 \leq i \leq n$ and hence that $concl(\iota)\theta \prec mprem(\iota')$. □

Lemma 2. *let σ be the most general unifier of terms s and s', and θ be any unifier of the same terms. Then for any term t, $(t\sigma)\theta = t\theta$.*

Proof. Since σ is the most general unifier, there must be a substitution ρ such that $\sigma\rho = \theta$. Hence $(t\sigma)\theta = (t\sigma)\sigma\rho = t\sigma\rho = t\theta$ where the second to last step follows from the fact that σ is idempotent. □

Lemma 3. *For every clause set N saturated by Inf, there exists a $gsel \in \mathcal{G}(sel)$ such that $GInf^{gsel}\ (\mathcal{G}(N)) \subseteq \mathcal{G}^{gsel}(Inf(N)) \cup GRed_I^{gsel}(\mathcal{G}(N))$.*

Proof. For every $D \in \mathcal{G}(N)$ there must exist a clause $C \in N$ such that $D \in \mathcal{G}(C)$. Let $\gg$ be an arbitrary well-founded ordering on clauses. We let $C = \mathcal{G}^{-1}(D)$ denote the $\gg$-smallest clause such that $D \in \mathcal{G}(C)$. We then choose the $gsel \in \mathcal{G}(sel)$ that for a clause $D \in \mathcal{G}(N)$ selects the corresponding literals to those selected by sel in $\mathcal{G}^{-1}(D)$. Given this $gsel$, we need to show that every inference with premises in $\mathcal{G}(N)$ is either the ground instance of an inference with premises in N, or is redundant with respect to $\mathcal{G}(N)$.

A Sup inference is redundant if the term t replaced in the second premise occurs at or below a variable. The proof is exactly the same as in the standard proof of the completeness of superposition [3], so we don't repeat it. All other inferences can be shown to be the ground instance of inferences from clauses in N.

Let $\iota \in GInf^{gsel}$ be the following GSup inference with premises in $\mathcal{G}(N)$.

$$\frac{D'\theta \vee t\theta \approx t'\theta \qquad C'\theta \vee s\theta[t\theta] \dot{\approx} s'\theta}{C'\theta \vee D'\theta \vee s\theta[t'\theta] \dot{\approx} s'\theta}$$

where $\mathcal{G}^{-1}(D\theta) = D = D' \vee t \approx t'$, $\mathcal{G}^{-1}(C\theta) = C = C' \vee s \dot{\approx} s'$ and ι fulfils all the side conditions of GSup. Let σ be any substitution. The literal $t\theta \approx t'\theta$ being strictly maximal in $D\theta$ implies that $t\sigma \approx t'\sigma$ is strictly maximal in $D\sigma$ due to the stability under substitution of $\succ$. The literal $s\theta[t\theta] \dot{\approx} s'\theta$ being (strictly) eligible in $C\theta$ with respect to $gsel$ implies that $s\sigma \approx s'\sigma$ is strictly eligible in $C\sigma$ with respect to sel. Let p be the position of $t\theta$ within $s\theta$ and let u be the subterm of s at p. Since the term $t\theta$ does not occur below a variable of C, such a position must exist. Moreover, u cannot be a variable since if it was $t\theta$ would occur at a variable of C. As θ is a unifier of u and t, it must be the case that either t is a variable, or u and t have the same top symbol. Further, $D\theta \prec C\theta$ implies that $C\sigma \not\preceq D\sigma$, $t\theta \succ t'\theta$ implies that $t\sigma \not\preceq t'\sigma$, and $s\theta[t'\theta] \succ s'\theta$ implies $s\sigma \not\preceq s'\sigma$. Thus, if t is not a variable, there exists the following Sup inference ι' from clauses D and C.

$$\frac{D' \vee t \approx t' \qquad C' \vee s[u] \dot{\approx} s'}{C' \vee D' \vee s[t'] \dot{\approx} s' \vee \mathcal{CS}}$$

We have that $(C' \vee D' \vee s[t'] \dot{\approx} s')\theta = \mathit{concl}(\iota)$. That is, the grounding of the conclusion of ι' less the constraint literals is equal to the conclusion of ι. Thus, ι is the θ-ground instance of ι' as per Definition 1. If t is a variable x, then there exists the following VSup inference ι' from clauses D and C.

$$\frac{D' \vee x \approx t' \qquad C' \vee s[u] \dot{\approx} s'}{(C' \vee D' \vee s[t'] \dot{\approx} s')\sigma}$$

where $\sigma = \{x \to u\}$ is the most general unifier of t and u. Thus, we can use Lemma 2 to show that $\mathit{concl}(\iota')\theta = \mathit{concl}(\iota)$ and again ι is the θ-ground instance of ι'.

Let $\iota \in \mathit{GInf}^{gsel}$ be the following GEqFact inference with premise in $\mathcal{G}(N)$.

$$\frac{C'\theta \vee u'\theta \approx v'\theta \vee u\theta \approx v\theta}{C'\theta \vee v\theta \not\approx v'\theta \vee u\theta \approx v\theta}$$

where $u'\theta = u\theta$, $\mathcal{G}^{-1}(C\theta) = C = C' \vee u' \approx v' \vee u \approx v$ and ι fulfils all the side conditions of GEqFact. Let σ be any substitution. The literal $u\theta \approx v\theta$ being maximal in $D\theta$ implies that $u\sigma \approx v\sigma$ is maximal in $D\sigma$. Since θ is a unifier of u' and u, at least one of them must be a variable, or they must share a top symbol. Moreover, $u\theta \succ v\theta$ implies that $u\sigma \not\preceq v\sigma$ and $u'\theta \succ v'\theta$ implies that $u'\sigma \not\preceq v'\sigma$. If neither u nor u' is a variable, there exists the following EqFact inference ι' from C.

$$\frac{C' \vee u' \approx v' \vee u \approx v}{C' \vee v \not\approx v' \vee u \approx v \vee \mathcal{CS}}$$

We have $(C' \vee v \not\approx v' \vee u \approx v)\theta = \mathit{concl}(\iota)$, making ι the θ-ground instance of ι' as per Definition 1. If either u of $'u$ is a variable there exists the following VEqFact inference ι' from C.

$$\frac{C' \vee u' \approx v' \vee u \approx v}{(C' \vee v \not\approx v' \vee u \approx v)\sigma}$$

where σ is the most general unifier of u and u'. Thus, we can use Lemma 2 to show that $\mathit{concl}(\iota')\theta = \mathit{concl}(\iota)$. Finally, let $\iota \in \mathit{GInf}^{gsel}$ be the following GEqRes inference with premise in $\mathcal{G}(N)$.

$$\frac{C'\theta \vee s\theta \not\approx s'\theta}{C'\theta}$$

where $s\theta = s'\theta$, $\mathcal{G}^{-1}(C\theta) = C = C' \vee s \not\approx s'$ and ι fulfils all the side conditions of GEqRes. Let σ be any substitution. The literal $s\theta \not\approx s'\theta$ being eligible with respect to gsel in $C\theta$ implies that $s \not\approx s'$ is eligible in C with respect to sel. Since θ is a unifier of s and s', at least one of them must be a variable, or they must share a top symbol. If $s = s'$, then there exists the following ReflDel inference ι' from C.

$$\frac{C' \vee s \not\approx s}{C'}$$

Otherwise we have two options. If either s (or analogously s') is a variable, then there is the following Bind inference ι' from C.

$$\frac{C' \vee x \not\approx s'}{C'\sigma}$$

Otherwise s and s' must share a top symbol and there is the following Decompose inference ι' from C.

$$\frac{C' \vee f(\overline{s}_n) \not\approx f(\bar{t}_n)}{C' \vee \mathcal{CS}}$$

In the first case, we have $concl(\iota')\theta = concl(\iota)$. In the second case, σ is the most general unifier of s and s', so we can use Lemma 2 to show that $concl(\iota')\theta = concl(\iota)$. In the last case, we have that $C'\theta = concl(\iota)$. Thus in all cases, ι is the θ-ground instance of ι'. □

Using Lemmas 1 and 3 we can instantiate Theorem 14 to prove the static refutational completeness of *Inf*. There is a slight issue here, as Theorem 14 gives us refutational completeness with respect to Herbrand entailment. That is $N \models M$ if $\mathcal{G}(N) \models \mathcal{G}(M)$. We would like to prove completeness with respect to entailment as defined in Sect. 2 (known as Tarski entailment). This issue can easily be resolved by showing that the two concepts are equivalent with regards to refutations which can be achieved in a manner similar to Bentkamp et al. (Lemma 4.19 of [6]).

Theorem 1 (Static refutational completeness). *For a set of clauses N saturated up to redundancy by Inf, $N \models \bot$ if and only if $\bot \in N$.*

Theorem 17 of Waldmann et al.'s framework can be used to derive dynamic refutational completeness from static refutational completeness. We refer readers to the framework for the formal definition of dynamic refutational completeness.

Theorem 2 (Dynamic refutational completeness). *The inference system Inf is dynamically refutationally complete with respect to the redundancy criterion (Red_I, Red_{Cl}).*

6 Extending to Higher-Order Logic

We sketch how the ideas above can be extended to higher-order logic. This is ongoing research, and many of the technical details have yet to be fully worked out. Here, we provide a (very) informal description and then provide examples. The higher-order unification problem is undecidable and there can exist a potentially infinite number of incomparable most general unifiers for a pair of terms [12]. Existing higher-order paramodulation style calculi deal with this issue in two main ways. One method is to abandon completeness and only unify to some predefined depth [22]. Another approach is to produce potentially infinite streams of unifiers and interleave the fetching of items from such streams with the standard saturation procedure [7]. Our idea is to solve easy sub-problems eagerly, such as when terms are first-order or in the pattern fragment [16], and add harder sub-problems as constraints. We then utilise dedicated inferences on negative literals to mimic the rules of Huet's well known (pre-)unification procedure [12]. We think that inferences similar to the following two, could be sufficient to achieve refutational completeness.

$$\frac{C' \vee x\,\overline{s}_n \not\approx f\,\overline{t}_m}{(C' \vee x\,\overline{s}_n \not\approx f\,\overline{t}_m)\{x \rightarrow \lambda\overline{y}_n.\, f\,(z_1\,\overline{y}_n)\ldots(z_m\,\overline{y}_n)\}}\ \textsc{Imitate}$$

$$\frac{C' \vee x\,\overline{s}_n \not\approx f\,\overline{t}_m}{(C' \vee x\,\overline{s}_n \not\approx f\,\overline{t}_m)\{x \rightarrow \lambda\overline{y}_n.\, y_i\,(z_1\,\overline{y}_n)\ldots(z_p\,\overline{y}_n)\}}\ \textsc{Project}$$

In both rules, each z_i is a fresh variable of the relevant type, and $x\,\overline{s}_n \not\approx f\,\overline{t}_m$ is selected in C. Project has $k \leq n$ conclusions, one for each y_i of suitable type. We hope that through a careful definition of the selection function, along with the use of purification, we can avoid the need to apply unification inferences to flex-flex literals (negative literals where both sides of the equality have variable heads). Moreover, we are hopeful that the calculus we propose can remain complete without the need for inferences that carry out superposition beneath variables such as the FluidSup rule of λ-superposition [7] and the SubVarSup rule of combinatory-superposition [9].

Example 3. Consider the unsatisfiable clause set:

$$C_1 = f\,y\,(x\,a)\,(x\,b) \not\approx t \qquad C_2 = f\,c\,a\,b \approx t$$

A Sup inference between C_1 and C_2 results in clause $C_3 = t\sigma \not\approx t\sigma \vee x\,a \not\approx a \vee x\,b \not\approx b$ where $\sigma = \{y \rightarrow c\}$. Assume that the literal $x\,a$ is selected in C_3. We can carry out either a Project step on this literal or an Imitate step. The result of a project step is $C_4 = (t\sigma \not\approx t\sigma \vee (\lambda z.\, z)\,a \not\approx a \vee x\,b \not\approx b)\{x \rightarrow \lambda z.\, z\}$. Applying the substitution and β-reducing results in $C_5 = t\sigma \not\approx t\sigma \vee a \not\approx a \vee b \not\approx b$ from which it is easy to reach a contradiction.

Example 4 (Example 1 of Bentkamp et al. [7]*).* Consider the unsatisfiable clause set:

$$C_1 = f\,a \approx c \qquad C_2 = h\,(y\,b)\,(y\,a) \not\approx h\,(g\,(f\,b))\,(g\,c)$$

An EqRes inference on C_2 results in $C_3 = y\,b \not\approx g\,(f\,b) \vee y\,a \not\approx g\,c$. An Imitate inference on the first literal of C_3 followed by the application of the substitution and some β-reduction results in $C_4 = g\,(z\,b) \not\approx g\,(f\,b) \vee g\,(z\,a) \not\approx g\,c$. A further double application of EqRes gives us $C_5 = z\,b \not\approx f\,b \vee z\,a \not\approx c$. We again carry out Imitate on the first literal followed by an EqRes to leave us with $C_6 = x\,b \not\approx b \vee f\,(x\,a) \not\approx c$. We can now carry out a Sup inference between C_1 and C_6 resulting in $C_7 = x\,b \not\approx b \vee c \not\approx c \vee x\,a \not\approx a$ from which it is simple to derive $\bot$ via an application of Imitate on either the first or the third literal. Note, that the empty clause was derived without the need for an inference that simulates superposition underneath variables, unlike in [7].

Example 5 (Example 2 of Bentkamp et al. [7]*).* Consider the unsatisfiable clause set:

$$C_1 = f\,a \approx c \qquad C_2 = h\,(y\,(\lambda x.\,g\,(f\,x))\,a)\,y \not\approx h\,(g\,c)\,(\lambda w\,x.\,w\,x)$$

An EqRes inference on C_2 results in $C_3 = y\,(\lambda x.\,g\,(f\,x))\,a \not\approx g\,c \vee y \not\approx \lambda w\,x.\,w\,x$. Assuming that the second literal is selected,[2] an EqRes inference results in $C_4 = (y\,(\lambda x.\;\,g\,(f\,x))\,a \not\approx g\,c)\{y \rightarrow \lambda w\,x.\,w\,x\}$. Simplifying C_4 via applying the substitution and β-reducing, we achieve $g\,(f\,a) \not\approx g\,c$. Superposing C_1 onto this clause we end up with $C_5 = g\,c \not\approx g\,c$ from which the empty clause can easily be derived. Note again, that the empty clause has been derived without recourse to a FluidSup-like inference.

7 Experimental Results

We implemented the calculus in the Vampire theorem prover [14]. We also implemented a variant of the calculus, that utilises fingerprint indices [19] to act as an imperfect filter. The completeness proof indicates that a superposition inference only needs to be carried out when the two terms can *possibly* unify. Therefore, we store terms in fingerprint indices, which act as fast imperfect filters for finding unification partners, and only carry out superposition inferences with terms returned by the index. This restricts, somewhat, the number of inferences that take place, at the expense of some loss of speed. Thus, it represents a midway path between eager unification and delayed unification. As a final twist, we implemented a version of the calculus that uses fingerprint indices as well as solving constraint literals of the form $x \not\approx t$ (where x is not a subterm of t) and $t \not\approx t$ eagerly. Thus, in this version of the calculus there is no need for the Bind and ReflDel rules.

We compared each of these approaches with the standard superposition calculus implemented in Vampire. We refer to the standard calculus as Vampire and the delayed inference calculus without fingerprint indices by Vampire*.[3] We refer to the delayed inference calculus with fingerprint indices by Vampire†.

[2] Most orderings would select the first literal. In this case, we can still derive a contradiction, but the proof is longer.

[3] Our implementation can be found at https://github.com/vprover/vampire/tree/delayed-unification. To run the new calculus, use option `-duc on`. To run the standard calculus, the option `duc` is set to `off`.

Finally, we refer to the calculus that eagerly solves some constraint literals by Vampire‡.[4]

We tested these approaches against each other on benchmarks coming from CASC 2023 system competition [23]. As our new approach is not currently compatible with higher-order or polymorphic input, we restricted the comparison to monomorphic first-order problems. Namely, we used the 500 benchmarks in the FNE and FEQ categories. These are monomorphic, first-order benchmarks that either include equality (FEQ) or do not contain equality (FNE). All benchmarks in the set are theorems. The results can be seen in Table 1. All experiments were run on a node cluster located at The University of Manchester. Each node in the cluster is equipped with 192 gigabytes of RAM and 32 Intel® Xeon processors with two threads per core. Each configuration was given 100s of CPU time per problem and run in single core mode. Vampire was run with options `--mode casc` which causes it to use a tuned portfolio of strategies. All other variants were run with options `--mode casc --forced_options duc=on` which forces the use of the new calculus on top of the aforementioned portfolio.

Table 1. Summary of experimental results

Approach	Solved	Uniques
Vampire	430	110
Vampire*	238	0
Vampire†	255	0
Vampire‡	322	2

The calculi based on delayed unification perform badly in comparison to standard superposition. This is unsurprising, as syntactic first-order unification is already an efficient process. By replacing it with delayed unification, we gain little in terms of time, but pay a heavy penalty in terms of the number of inferences carried out. The use of fingerprint indices helps somewhat in mitigating this issue, but not a great deal. Eagerly solving trivial constraints shows more promise and is actually able to solve two problems that the standard calculus can not (within the time limit). These are the benchmarks `CSR036+3.p` and `LAT347+3.p`.

8 Related Work

The only other proof calculi that we are aware of that explicitly integrate unification rules at the calculus level, are the higher-order paramodulation calculi [8,22]

[4] The code for both Vampire† and Vampire‡ can be found at branch https://github.com/vprover/vampire/tree/delayed-unif-with-fp. Vampire† was built from commit `c04a08feb5db3e7468a1fa` and Vampire‡ from commit `fa2f139302b6a7a6487e73`. Again, option `-duc on` is required for the new calculi to run.

and lazy paramodulation [21]. However, these calculi are paramodulation calculi and do not incorporate certain concepts of redundancy so crucial to the success of superposition provers. Moreover, the completeness proofs for these calculi are based on very different techniques to the Bachmair & Ganzinger style model building proofs commonly employed in the completeness proofs of superposition calculi.

There are other calculi that in some form do represent the folding of unification into the calculus, but the link between the unification rules and the calculus is less clear. For example, the recent work by one of the authors of this paper [13] relating to reasoning about linear arithmetic, moves theory reasoning relating to a number of equations from the unification algorithm to the calculus level. A different example, by another of this paper, is the combinatory-superposition calculus [9] which essentially folds higher-order combinatory unification into the calculus. In both cases, the relationship between the unification algorithm and the calculus rules is not obvious.

There are other methods of dovetailing unification with inference rules. For example, a unification procedure can be modified to return a stream of results. This stream can be interrupted in order to carry out further inferences and then returned to later. This is the approach taken by the higher-order Zipperposition prover [7] in order to handle the infinite sets of unifiers returned by higher-order unification. Conceptually, this is a very different solution to using constraints, since the intermediate terms created during unification are not available to the entire calculus as they are in our approach. Furthermore, from an implementation perspective, streams of unifiers are a far greater departure from the standard saturation architecture than the adding of constraints. Unification can also be partially delayed by preprocessing techniques such as Brand's modification method and its developments [5].

As mentioned in the introduction, abstraction resembles the basic strategy [4,15], where unification problems are added to the constraint part of a clause. Periodically, these constraints can be checked for satisfiability and clauses with unsatisfiable constraints removed. However, in the basic strategy, the constraints do not interact with the rest of the proof calculus. Moreover, redundancy of clauses can no longer be defined in terms of ground instances, but only in terms of ground instances that satisfy the constraints. This significantly affects the simplification machinery of superposition/resolution.

Unification with abstraction was first introduced, to the best of our knowledge, by Reger et al. in [17] in the context of theory reasoning. However, the concept was introduced in an ad-hoc fashion with no theoretical analysis of its impact on the completeness of the underlying calculus. Recently, the relationship between unification modulo an equational theory and unification with abstraction has been analysed [13] and a framework developed linking the two. It remains to explore whether the current work can fit into that framework.

9 Conclusion

We have developed a first-order superposition calculus that delays unification through the use of constraints, and proved its completeness. Whilst the calculus does not perform well in practice, we feel that the calculus and its completeness proof form a template that can be followed to prove the completeness of calculi that involve unification procedures more complex than syntactic first-order unification. For example unification modulo a set of equations E. Some of the crucial features of our approach are: (1) the carrying out of partial unification and adding the remaining unification pairs back as constraints, and (2) the ignoring of constraint literals in the definition of redundant inference. In particular, feature (1) may well be crucial in taming issues relating to undecidable unification problems. For example, in higher-order logic where unification is undecidable, it is common to run unification to a particular depth and then give up if termination has not occurred. Of course, this harms completeness. With our approach it should be possible to add the remaining unification pairs back as constraints and maintain completeness. In the future, we would like to generalise our approach into a framework that can be used to prove the completeness of a variety of calculi as long as the unification problem for the underlying terms meets certain conditions. We would also like to explore instantiating such a framework to prove the completeness of particular calculi of interest to us such as AC-superposition and higher-order superposition.

Acknowledgements. We acknowledge funding from the ERC Consolidator Grant ARTIST 101002685, the TU Wien Doctoral College SecInt, and the FWF SFB project SpyCoDe F8504.

References

1. Baader, F., Snyder, W.: Unification theory. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. II, chap. 8, pp. 447–533 (2001)
2. Bachmair, L., Ganzinger, H.: On restrictions of ordered paramodulation with simplification. In: Stickel, M.E. (ed.) CADE 1990. LNCS, vol. 449, pp. 427–441. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52885-7_105
3. Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. J. Log. Comput. **4**(3), 217–247 (1994)
4. Bachmair, L., Ganzinger, H., Lynch, C., Snyder, W.: Basic paramodulation and superposition. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 462–476. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_185
5. Bachmair, L., Ganzinger, H., Voronkov, A.: Elimination of equality via transformation with ordering constraints (1997)
6. Bentkamp, A., Blanchette, J., Cruanes, S., Waldmann, U.: Superposition for lambda-free higher-order logic. Logical Methods Comput. Sci. **17** (2021)
7. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P., Waldmann, U.: Superposition with lambdas. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 55–73. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_4
8. Benzmüller, C., Sultana, N., Paulson, L.C., Theiß, F.: The higher-order prover LEO-II. J. Autom. Reason. **55**(4), 389–404 (2015)
9. Bhayat, A., Reger, G.: A combinator-based superposition calculus for higher-order logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12166, pp. 278–296. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_16

10. Domenjoud, E.: A technical note on AC-unification. The number of minimal unifiers of the equation $\alpha x_1 + \cdots + \alpha x_p \doteq_{AC} \beta y_1 + \cdots + \beta y_q$. J. Autom. Reason. **8** (1992)
11. Hoder, K., Voronkov, A.: Comparing unification algorithms in first-order theorem proving. In: Mertsching, B., Hund, M., Aziz, Z. (eds.) KI 2009. LNCS (LNAI), vol. 5803, pp. 435–443. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04617-9_55
12. Huet, G.P.: A unification algorithm for typed λ-calculus. Theor. Comput. Sci. **1**(1), 27–57 (1975)
13. Korovin, K., Kovács, L., Reger, G., Schoisswohl, J., Voronkov, A.: ALASCA: reasoning in quantified linear arithmetic. In: Sankaranarayanan, S., Sharygina, N. (eds.) TACAS 2023. LNCS, vol. 13993, pp. 647–665. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30823-9_33
14. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
15. Nieuwenhuis, R., Rubio, A.: Basic superposition is complete. In: Krieg-Brückner, B. (ed.) ESOP 1992. LNCS, vol. 582, pp. 371–389. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55253-7_22
16. Nipkow, T.: Functional unification of higher-order patterns. In: LICS, pp. 64–74. IEEE Computer Society (1993)
17. Reger, G., Suda, M., Voronkov, A.: Unification with abstraction and theory instantiation in saturation-based reasoning. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 3–22. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_1
18. Schulz, S.: E - a brainiac theorem prover. AI Commun. **15**(2,3), 111–126 (2002)
19. Schulz, S.: Fingerprint indexing for paramodulation and rewriting. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 477–483. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_37
20. Snyder, W., Gallier, J.: Higher-order unification revisited: complete sets of transformations. J. Symb. Comput. **8**(1–2), 101–140 (1989)
21. Snyder, W., Lynch, C.: Goal directed strategies for paramodulation. In: Book, R.V. (ed.) RTA 1991. LNCS, vol. 488, pp. 150–161. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-53904-2_93
22. Steen, A., Benzmüller, C.: The higher-order prover Leo-III. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 108–116. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_8
23. Sutcliffe, G., Suttner, C.: The state of CASC. AI Commun. **19**(1), 35–48 (2006)
24. Voronkov, A.: AVATAR: the architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 696–710. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_46
25. Waldmann, U., Tourret, S., Robillard, S., Blanchette, J.: A comprehensive framework for saturation theorem proving. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12166, pp. 316–334. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_18

On Incremental Pre-processing for SMT

Nikolaj Bjørner[1](✉) and Katalin Fazekas[2]

[1] Microsoft Research, Redmond, USA
nbjorner@microsoft.com
[2] TU Wien, Vienna, Austria
katalin.fazekas@tuwien.ac.at

Abstract. We introduce a calculus for incremental pre-processing for SMT and instantiate it in the context of z3. It identifies when powerful formula simplifications can be retained when adding new constraints. Use cases that could not be solved in incremental mode can now be solved incrementally thanks to the availability of pre-processing. Our approach admits a class of transformations that preserve satisfiability, but not equivalence. We establish a taxonomy of pre-processing techniques that distinguishes cases where new constraints are modified or constraints previously added have to be replayed. We then justify the soundness of the proposed incremental pre-processing calculus.

1 Introduction

Pre-processing is a central ingredient for scaling automated deduction. These techniques apply targeted global simplification steps that can drastically reduce the complexity of problems before search techniques that use mainly local inference steps are invoked. They are used across several solver domains, spanning SAT, to SMT, first-order automated theorem proving, constraint programming, and integer programming. With the exception of SAT solvers, prior techniques do not combine well when new constraints are added incrementally to a pre-processed state. Solvers have the option to restart pre-processing from scratch. This model is viable if the overall number of solver calls is small compared to time spent solving, but is not practical for scenarios where many minor variations of a set of main constraints are queried. Such scenarios may be found in applications of dynamic symbolic execution or symbolic model checking.

A procedure to incorporate pre- and in-processing techniques [27] into incremental SAT solvers was introduced in [18], where such incremental in-processing allowed a dramatic improvement in the performance of bounded model checking applications. In the case of SAT, the effect of a simplification step is recorded in a *reconstruction stack*. Each eliminated clause is saved on that stack together with a partial assignment, called its *witness*, that is used to show the redundancy of the eliminated clause. For example, the redundancy of blocked clauses are witnessed by their blocked literal, a literal that upon all resolvents are tautological [26,32]. The reconstruction stack has two very important roles in SAT

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 41–60, 2023.
https://doi.org/10.1007/978-3-031-38499-8_3

solvers. First of all, it has all the information that is necessary for model reconstruction [25]. When the elimination of a clause is not model-preserving, its witness on the stack tells how to modify or extend any found solution of the simplified formula such that it then satisfies the removed clause as well. Beyond that, the reconstruction stack allows to recognize all those previous simplification steps that are potentially invalidated by an incrementally added new constraint. For example, literals that were blocked in the global state of the previous clauses might not be blocked any more in the presence of some new constraints. Finding these clauses and their cone of influence on the reconstruction stack allows to *undo* only the problematic previous simplification steps, thereby allows pre- and in-processing to be incremental [18].

Motivated by incremental in-processing SAT solvers, our goal here is to pave a path towards a similar mechanism in the context of SMT solvers. However, SMT problems extend propositional SAT formulas in several dimensions: the base theory of SMT is the theory of equality over uninterpreted functions and predicates, SMT formulas may contain quantifiers, and constants and functions that have interpretations over theories. Concrete cases of incremental SMT pre-processing was considered in [19]. While most of the formula simplification techniques of SAT solvers are captured by well studied redundancy properties [23], such a unified understanding and description of SMT pre-processing techniques is not yet introduced. Though some redundancy notions of SAT solvers can be directly embedded or generalized to SMT [30], a notion that appears to capture simplifications in SMT in many cases is that of a *substitution*: an uninterpreted constant or function is defined into a solved form and the constraints are simplified based on the solution. When new constraints, containing the solved function symbols, are added after pre-processing, our method distinguishes between simplifications that allow applying the substitution to the new formula or removing the substitution and re-adding the old constraints that were simplified. We have found it useful to characterize pre-processing simplifications by the following categories.

Equivalence Preserving Simplifications. Many simplification methods are based on equivalence preserving simplifications. For example $x > x - y + 1$ simplifies to $y > 1$. They are automatically incremental by virtue of not changing the set of models. Developing equivalence preserving simplifications is a significant area of research and engineering by itself. A good example is using and-inverter graphs (AIGs) for simplifying propositional and first-order formulas [24,45]. The main challenge with developing equivalence preserving simplifications in an incremental setting is to make them efficient.

Rigid Constrained Simplifications. An important class of simplifications are based on eliminating variables by finding solutions to them. In the formula $x \leq y + 1 \wedge x \geq y + 1 \wedge \varphi[x, y]$ we can solve for x (or y) by setting $x \simeq y + 1$ and then substituting in the solution for x into φ. The simplified formula is $\varphi[y + 1, y]$. The set of models of the original formula must all satisfy the equality $x \simeq y + 1$. This property allows to reuse the simplification when later adding a formula $\psi[x, y]$. It can be added by applying the solution for x: $\psi[y + 1, y]$.

A model of $\varphi[y+1, y] \wedge \psi[y+1, y]$ must conversely correspond to a model of the original formulas $\varphi[x, y]$ and $\psi[x, y]$. The equality $x \mapsto y+1$ is used in a *model converter* to establish the original model. Some pre-processing techniques translate constraints from one domain to another. For example, formulas over bounded integers can be solved by translation into bit-vectors. This translation can be described with a set of equalities where bounded integers are solved for their bit-vector representation (see later an example in Table 1).

Under Constrained Simplifications. The rigid constrained simplifications already cover a significant class of pre-processing methods. Allowing incrementally solving for variables has a profound practical effect on using z3 incrementally in user scenarios. There is however a larger class of simplifications that also allow eliminating variables but do not preserve solutions to the eliminated variable. These simplifications have the same or more solutions for symbols in the original formula and we call them *under-constrained.* For example, the formula $((x \simeq y \wedge y < z+u) \vee y \geq z \cdot u)$ contains x in only one position. It can be replaced by the formula $((b \wedge y < z+u) \vee y \geq z \cdot u)$ where b is fresh. Similarly introducing definitions of fresh symbols does not eliminate solutions to symbols in the original formula. Lastly, when removing redundant clauses, the new formula may have more solutions. Tseitin transformation introduces definitions that allow removing redundant, non-CNF, formulas.

Over Constrained Simplifications. Symmetry reduction [14,38] and strengthening using propagation redundancy criteria [37] are prominent examples of simplifications that apply strengthening to reduce the search space. These transformations are not covered by the classes covered by our main result. We leave it to future work to examine whether or how to incorporate strengthening: one avenue is to leverage assumption literals [16] to temporarily enable strengthenings either as part of pre-processing or during search [39].

Table 1 summarizes the main categories of pre-processing techniques discussed so far. This paper develops a calculus of incremental pre-processing for rigid constrained, under-constrained, clause elimination, and introduction of definitions. However, it does not discuss further over-constrained simplifications.

In this paper we introduce the concept of *simplification modulo substitutions* and show that the main SMT pre-processing methods maintain such a property. Based on that, we show how to apply or revert the effect of previous pre-processing steps when new formulas are added after simplification.

2 Preliminaries

We assume the usual notions of first-order logic with equality, satisfiability, logical consequence and theory, as described e.g. in [17]. An interpretation $\mathcal{M}$ for a signature Σ (or Σ-model) consists of a non-empty set $\mathcal{U}_{\mathcal{M}}$ called the universe of the model, and a mapping $(\_)^{\mathcal{M}}$ assigning to each variable and constant symbol an element of $\mathcal{U}_{\mathcal{M}}$, to each n-ary function symbol f in Σ an n-ary function $f^{\mathcal{M}}$

Table 1. Main categories of pre-processing techniques found in SMT solvers. Function *ite* is an abbreviation for *if-then-else* and *bv2int* is a function that maps a bit-vector to an integer value.

category	example input	example output	model converter
equivalence	$x > x - y + 1$	$y > 1$	ε
rigid	$x \simeq t, \varphi$	$\varphi[t/x]$	$x \mapsto t$
	$0 \leq x \leq 1 \wedge (x \simeq 1 \vee y > 0)$	$x_b \vee y > 0$	$x \mapsto \mathit{ite}(x_b, 1, 0)$
	$1 \leq x \leq 4 \wedge (x \simeq 1 \vee y > 0)$	$b_{[2]} \simeq 0 \vee y > 0$	$x \mapsto 1 + \mathit{bv2int}(b_{[2]})$
under	$F, ((x \simeq t \wedge \varphi) \vee \psi)$ $x \notin FV(\psi), FV(F)$	$F, (\varphi[t/x] \vee \psi)$	$x \mapsto t$
	$F, x \leq y, x \leq z, y \leq u$ $x, y \notin FV(F)$	F	$x \mapsto \min(y, z), y \mapsto u$
def-intro	$(a \wedge b) \vee c$	$\neg x_b \vee a, \neg x_b \vee b, x_b \vee c$	ε
redundant	$F, \neg p \vee \neg q, p \vee q$ p is positive in F	$F, \neg p \vee \neg q$	$p \mapsto p \vee \neg q$
over	$p(x), p(y), p(z)$	$x \leq y \leq z$ $p(x), p(y), p(z)$	ε

from $\mathcal{U}_{\mathcal{M}}^n$ to $\mathcal{U}_{\mathcal{M}}$, and to each n-ary predicate symbol p in Σ an n-ary function from the set $\mathcal{U}_{\mathcal{M}}^n$ to distinguished values representing true and false. Note that to keep the presentation simple, we only consider a single universe in the models. Interpretations extend to terms by composition.

We use the terminology *symbols* referring to uninterpreted symbols (variables) and function symbols. Given a model $\mathcal{M}$ and a symbol x, the model $\mathcal{M}[x \mapsto a]$ is exactly the same as $\mathcal{M}$, except that $x^{\mathcal{M}} = a$ where $a \in \mathcal{U}_{\mathcal{M}}$ for 0-ary symbols and a is a function over $\mathcal{U}_{\mathcal{M}}$ for n-ary function or predicate symbols.

Lemma 1 (Translation Lemma [41]**).** *If F is a formula and t is a term s.t. no variable in t occurs bound in F, then $\mathcal{M} \models F[t/x]$ iff $\mathcal{M}[x \mapsto t^{\mathcal{M}}] \models F$.*

Note that we may use λ terms to represent updates to function and predicate symbols. The interpretation of a λ term is a function.

We denote *Skolem symbols* for n-ary functions (where $n = 0$ is possible) that cannot occur in input formulas. Only pre-processing methods may introduce the Skolem symbols as a guarantee that they are fresh.

Convention 1 (Variable non-capture). *Throughout this paper we assume that free and bound variables are disjoint, such that when we substitute a term t for a variable x in formula F, none of the variables in t are captured.*

Definition 1 (Labeled substitution). *$\langle x \leftarrow t; \Psi \rangle^{\mathbb{B}}$ represents a substitution of x by t, justified by the formula Ψ. The label $\mathbb{B}$ is either $\top$ or $\bot$ and it indicates whether the map $x \mapsto t$ may be used as an equal replacement of Ψ.*

Example 1. The labeled substitution $\langle x \leftarrow y + 1; x \simeq y + 1 \rangle^{\bot}$ represents the substitution of x by $y + 1$ justified by the formula $x \simeq y + 1$. The label $\bot$ of the substitution indicates that applying the substitution on a formula F where $x \simeq y + 1$ is present does not change the set of models of the formula.

Definition 2. *Given $\theta = \langle x_1 \leftarrow t_1; \Psi_1\rangle^{\mathbb{B}_1} \langle x_2 \leftarrow t_2; \Psi_2\rangle^{\mathbb{B}_2} \dots \langle x_n \leftarrow t_n; \Psi_n\rangle^{\mathbb{B}_n}$ and an interpretation $\mathcal{M}$, we define the interpretation $\mathcal{M}\theta$ as follows:*

$$\mathcal{M}\varepsilon = \mathcal{M}$$
$$\mathcal{M}\theta\langle x \leftarrow t; \Psi\rangle^{\mathbb{B}} = (\mathcal{M}[x \mapsto t^{\mathcal{M}}])\theta$$

Definition 3. *Given $\theta = \langle x_1 \leftarrow t_1; \Psi_1\rangle^{\mathbb{B}_1} \langle x_2 \leftarrow t_2; \Psi_2\rangle^{\mathbb{B}_2} \dots \langle x_n \leftarrow t_n; \Psi_n\rangle^{\mathbb{B}_n}$ and a formula F, we define the formula $F\theta$ as follows:*

$$F\varepsilon = F$$
$$F\langle x \leftarrow t; \Psi\rangle^{\mathbb{B}}\theta = (F[t/x])\theta$$

Informally, a sequence of substitutions θ is applied to interpretations from right to left (i.e. backwards), while to formulas from left to right (i.e. forward). Further, note that the translation lemma generalizes in a straight-forward way to substitutions.

3 Incremental Pre-processing

In this section we introduce a calculus to describe incremental pre-processing for SMT based on the following notion.

Definition 4 (Simplification modulo θ). *We say that the formula F simplifies to F' modulo θ, denoted $F \succeq_\theta F'$ if*

- *If $\mathcal{M} \models F$ then there is a model $\mathcal{M}'$ such that, $\mathcal{M}' \models F'$ and $\mathcal{M}'$ agrees with $\mathcal{M}$ on all symbols that are in F or in background theories or not in F'.*
- *If $\mathcal{M}' \models F'$ then $\mathcal{M}'\theta \models F$.*

It follows that simplification allows transitive chaining assuming that symbols are not recycled.

Lemma 2 (Transitivity of simplification). *Let $F \succeq_\theta F'$ and $F' \succeq_{\theta'} F''$ such that every symbol that is both in F and F'' also occurs in F' (i.e. old symbols are not re-introduced). Then $F \succeq_{\theta\theta'} F''$.*

3.1 Simplification Rules

There are several possible situations where the concept of simplification modulo substitutions can be used to capture potential simplification steps. For example, a useful special case for simplification modulo θ is when a formula F implies an equality $x \simeq t$ that can then be turned into a substitution to simplify F.

Example 2. The formula $isCons(x) \wedge F[x]$ implies $\exists h, t \;.\; x \simeq cons(h, t)$, where h, t are fresh variables (corresponding to the head and tail of a cons list). We may substitute x by $cons(h, t)$ in $F[x]$ to eliminate x. The literal $isCons(cons(h, t))$ is equivalent true and $F[cons(h, t)]$ is a model simplification of the original formula modulo $x \simeq cons(h, t)$.

There are also useful special cases where a formula F *does not* imply an equality $x \simeq t$, but the same equality may still be used to simplify F.

Example 3. In the formula $F := ((x \simeq 3 \wedge x > u) \vee y > u) \wedge u > z$ we can substitute $x \mapsto 3$ and retain simplification. The formula F simplifies to $F[3/x] := (3 > u \vee y > u) \wedge u > z$, but F does not imply $x = 3$.

There are also cases where substitutions are not suitable to describe the relation between F and F'. It is easier to characterize these by the property that F' is a proper subset of F.

Example 4. A blocked clause $p \vee C$ can be removed from a set of formulas without changing satisfiability: $F, (p \vee C) \succeq_{p \mapsto p \vee \neg C} F$. If we were to substitute p by $p \vee \neg C$ everywhere in F it would weaken clauses where p occurs positively.

Finally, it is possible to accomodate cases where pre-processing *introduces* definitions, such as through the *unfold* transformation (see Sect. 6.5), or by Skolemization and Tseitin transformations.

Example 5. The Skolemization of $\forall x \;.\; \exists y \;.\; p(x, y)$ is $\forall x \;.\; p(x, f_{sk}(x))$. Here the original quantified formula is replaced by the Skolemized formula.

We model the pre-processing performed by an SMT solver as a sequence of abstract states where each state consists of two components: a formula F and an ordered sequence of labeled substitutions θ. Based on the shown cases, we formulate the following conditions for applying simplification rules in Fig. 1.

Rigid :

$$F \parallel \theta \Longrightarrow F[t/x] \parallel \theta\langle x \leftarrow t; \Psi\rangle^{\perp} \quad \text{if } \Psi \subseteq F, x \not\in t, \text{ and } \Psi \Rightarrow \exists y \;.\; x \simeq t[y]$$

Flex :

$$F, \Psi \parallel \theta \Longrightarrow F, \Psi[t/x] \parallel \theta\langle x \leftarrow t; \Psi\rangle^{\top} \quad \text{if } x \in \Psi, x \not\in F \text{ and } \Psi \succeq_{x \mapsto t} \Psi[t/x]$$

Update :

$$F, \Psi \parallel \theta \Longrightarrow F, \Phi \parallel \theta\langle x \leftarrow t; \Psi\rangle^{\top} \quad \text{if } F, \Psi \succeq_{x \mapsto t} F, \Phi$$

Fig. 1. A calculus for pre-processing in SMT

We formulated the side conditions that allow to identify a minimal set of conjuncts Ψ of F involved with the solution for x. Note that a simplification remains valid when adding conjuncts that do not contain x. The Update rule handles broadly a set of simplifications, including proof rules from DRAT systems and introduction of definitions and Skolemization. It may be presented in forms where Φ or Ψ or the substitution are empty. The substitution $x \mapsto t$ generally represents a tuple of symbols x replaced by terms t. To simplify presentation we only discuss the case where x is a single symbol and we elide rules that preserve equivalence. The Update rule records Ψ so it can later be re-added in case a new constraint mentions x. This may be overkill when $\Phi[t/y] = \Psi$ for y fresh (in Sect. 4 we will show another rule, Invert, that adds only the equality $y \simeq t$ in such cases).

Lemma 3. *If $F \Rightarrow \exists y\ .\ x \simeq t[y]$, s.t. $y \notin F$, $x \notin t$, and t is substitutable for x in F, then $F \succeq_{x \mapsto t} F[t[y]/x]$.*

Proof. Let $\mathcal{M}$ be an interpretation s.t. $\mathcal{M} \models F$. Then $\mathcal{M} \models F \wedge \exists y\ .\ x \simeq t[y]$ and by definition of the satisfaction relation, there must exists an $a \in \mathcal{U}_{\mathcal{M}}$, s.t. $\mathcal{M}[y \mapsto a] \models F \wedge x \simeq t[y]$. Let $\mathcal{M}'$ note $\mathcal{M}[y \mapsto a]$. From $\mathcal{M}' \models F \wedge x \simeq t[y]$ follows that $x^{\mathcal{M}'} = t[y]^{\mathcal{M}'}$ and so $F^{\mathcal{M}'} = F[t[y]/x]^{\mathcal{M}'}$. Since $\mathcal{M}' \models F$, we have that $\mathcal{M}' \models F[t[y]/x]$. For the other direction, when $\mathcal{M}' \models F[t[y]/x]$, due to Lemma 1, $\mathcal{M}'[x \mapsto t[y]^{\mathcal{M}'}] \models F$. Hence, $F \succeq_{x \mapsto t} F[t[y]/x]$. □

Corollary 1. *The side-condition for* Rigid *implies that $F \succeq_{x \mapsto t} F[t/x]$.*

Lemma 4. *Assume $\Psi \subseteq F, x \notin F \setminus \Psi$ and $\Psi \succeq_{x \mapsto t} \Psi[t/x]$, then $F \succeq_{x \mapsto t} F[t/x]$.*

Proof. Since $x \notin F$, $(F \setminus \Psi) = (F \setminus \Psi)[t/x]$, thus $(F \setminus \Psi) \succeq_{x \mapsto t} (F \setminus \Psi)[t/x]$. Then, from $\Psi \succeq_{x \mapsto t} \Psi[t/x]$ follows that $F \succeq_{x \mapsto t} F[t/x]$.

Lemma 3 established that the side-condition for Rigid ensures simplification modulo θ. We therefore have the following corollaries.

Corollary 2. *If a formula F' is derived from F by the inferences from Fig. 1, then it has the property $F \succeq_{x \mapsto t} F'$.*

The other rules enforce preservation of satisfiability in their side-conditions.

Corollary 3. *The rules from Fig. 1 preserve satisfiability.*

The transitive application of the simplifications also preserve satisfiability in a way that extends the notion of simplification modulo a substitution.

Proposition 1. *Consider a formula F_0 and a state $F \parallel \theta$ derived from $F_0 \parallel \varepsilon$ using the rules from Fig. 1. Then $F_0 \succeq_\theta F$.*

Proof. It follows as Corollary 2 notes that each application of a rule from Fig. 1 is a simplification modulo and Lemma 2 notes that simplification modulo is transitive.

Informally, Proposition 1 means that using θ, one can transform any model of the simplified formula into a model of the original input formula. Note that the simplified F may contain fresh Skolem symbols that are not occurring in F_0.

3.2 Pre-processing Replay

Rules of Fig. 1 captured possible pre-processing steps that can be applied on a single SMT problem. We now describe the scenario where we add additional constraints Φ to a pre-processed state. Without incremental pre-processing we have the option to conjoin Φ to the original formula F_0 and re-run pre-processing. The goal of incremental pre-processing is to retain as much of the effect of previous work as possible.

We will show that for pre-processing steps derived by rule Rigid it is possible to apply the corresponding substitution to Φ directly, while the other simplification steps may require to re-introduce formulas that were previously removed. We call this process of applying the effect of simplifications on a new formula as pre-processing *replay*. Figure 2 shows an imperative implementation of pre-processing replay.

```
Replay (formula $\Phi$, substitution sequence $\theta = \sigma_1, \ldots \sigma_n$ )
$\theta' := \langle\rangle$
for $\langle x_i \leftarrow t_i; \Psi_i\rangle^{\mathbb{B}_i}$ from $\sigma_1$ to $\sigma_n$
    if $x_i \in FV(\Phi)$ then
        if $\mathbb{B}_i = \top$ then // substitution is not Rigid
            $\Phi := \Phi \cup \Psi_i$ // re-introduce
        else
            $\Phi := \Phi[t_i/x_i]$ // apply
            $\theta' := \theta'\langle x_i \leftarrow t_i; \Psi_i\rangle^{\mathbb{B}_i}$
    else
        $\theta' := \theta'\langle x_i \leftarrow t_i; \Psi_i\rangle^{\mathbb{B}_i}$
return $\langle \Phi, \theta' \rangle$
```

Fig. 2. Algorithm Replay

Our main proposition summarizes the main property of Replay and ensures that an arbitrary formula Φ can be added mid-stream after pre-processing.

Proposition 2. *Let $F \parallel \theta$ be a state resulting from pre-processing F_0, and let $F \wedge \Phi' \parallel \theta'$ be a state produced by applying procedure Replay to Φ and θ, then $F_0 \wedge \Phi$ is equi-satisfiable to $F \wedge \Phi'$.*

To establish Proposition 2 we will introduce a calculus for reverting the effect of simplifications. It is shown in Fig. 3 and comprises of two rules, one for adding a formula with a substitution to F, the other both reverts the effect of a simplification and adds the reverted formula to F. The inferences rely on a side-condition that the formulas Φ, Ψ are *clean* relative to the substitution θ.

Definition 5. *A formula Φ is clean w.r.t. a substitution sequence θ iff*

$$\begin{array}{llll}
\textsc{Add}: & & & \\
F \parallel \theta & \Longrightarrow & F, \varPhi\theta \parallel \theta & \text{if } \varPhi \text{ is clean w.r.t. } \theta \\
\textsc{Undo}: & & & \\
F \parallel \theta_0 \langle x \leftarrow t; \varPsi \rangle^{\mathbb{B}} \theta & \Longrightarrow & F, \varPsi\theta \parallel \theta_0\theta & \text{if } \varPsi \text{ is clean w.r.t. } \theta
\end{array}$$

Fig. 3. A calculus for reverting pre-processing. Undo reverts a simplification by reintroducing a constraint. It prunes θ until Add applies for a new constraint $\varPhi$.

- *$\theta = \varepsilon$, or*
- *$\theta = \langle x \leftarrow t; \varPsi \rangle^{\mathbb{B}} \theta'$, $x \notin \varPhi$ and $\varPhi$ is clean with respect to θ', or*
- *$\theta = \langle x \leftarrow t; \varPsi \rangle^{\perp} \theta'$ and $\varPhi[t/x]$ is clean with respect to θ'.*

Thus, intuitively, $\varPhi$ is clean w.r.t. θ if $\varPhi\theta$ uses only Rigid substitutions from θ.

We now establish that formulas that are clean relative to θ can be added (after substitution) to formulas while maintaining models. The substitution used in rigid updates corresponds to equalities that are consequences.

Lemma 5. *Given a state $F' \parallel \theta\theta'$ derived from the state $F \parallel \theta$ and formula $\varPhi$ that is clean with respect to θ', then $F \wedge \varPhi \succeq_{\theta'} F' \wedge \varPhi\theta'$.*

Proof. We examine the two directions.

- Let $\mathcal{M} \models F \wedge \varPhi$. Induction on the length of the derivation from F to F' establishes that if $\mathcal{M} \models F$, then there is a corresponding $\mathcal{M}'$ such that $\mathcal{M}' \models F' \wedge \bigwedge_{(x \mapsto t) \in \theta'} x \simeq t$: Each time Rigid is applied a new equality is used for simplification $F_1[t_1/x_1]$. The equality can be added to the result, $F_1[t_1/x_1] \wedge x_1 \simeq t_1$ without changing satisfiability because x_1 does not occur in $F_1[t_1/x_1]$. Thus, the resulting model $\mathcal{M}'$ can be constrained to satisfy all equalities used in rigid substitutions. Since $\mathcal{M}' \models \varPhi$ already, then $\mathcal{M}' \models \varPhi\theta'$.
- Let $\mathcal{M}' \models F' \wedge \varPhi\theta'$. Then from the assumption of simplification modulo θ', we get $\mathcal{M}'\theta' \models F$. Lemma 1 ensures $\mathcal{M}'\theta' \models \varPhi$. Thus, $\mathcal{M}'\theta' \models F \wedge \varPhi$.

The correctness of the Add rule is now immediate:

Corollary 4. *Let $F \parallel \theta$ be derived from $F_0 \parallel \varepsilon$, and $\varPhi$ clean with respect to θ, then $F_0 \wedge \varPhi$ simplifies modulo θ to $F \wedge \varPhi\theta$.*

Proof. It follows from Lemma 5.

With Proposition 1 we established that Rigid, Flex and Update maintain $F_0 \succeq_\theta F$. We need to show that also for rule Undo. The first step is to establish that the formula removed by each of the pre-processing rules can be re-added without affecting simplification.

Lemma 6. *Given an inference $F \parallel \theta \Longrightarrow F' \parallel \theta\langle x \leftarrow t; \varPsi \rangle^{\mathbb{B}}$ by either of the rules Rigid, Update, Flex the formula F simplifies to $F', \varPsi$ modulo ε.*

Proof. The proof is by case analysis by the rule that is applied.

- Flex: Then $F' = F[t/x]$, $\Psi \subseteq F$ and therefore $F' \wedge \Psi = F \wedge \Psi[t/x]$. From the side condition $\Psi \succeq_{x \mapsto t} \Psi[t/x]$ every model of F there is a model of $\Psi[t/x]$ that agrees with symbols from F. Conversely F', Ψ properly contains F and therefore implies it. Therefore, $F \succeq_{\varepsilon} F', \Psi$.
- Update: We want to show that F, Ψ simplifies to F, Ψ, Φ modulo ε. The premise of Update ensures that for every $\mathcal{M} \models F, \Psi$ there is a model agreeing with $\mathcal{M}$ on symbols in F, Ψ, that satisfies F, Φ. Since interpretation of the symbols in Ψ is unchanged it also satisfies Ψ. Conversely, if $\mathcal{M}' \models F, \Psi, \Phi$, then already $\mathcal{M}' \models F, \Psi$ and therefore $\mathcal{M}'\varepsilon \models F, \Psi$.
- Rigid: We wish to establish that $F \succeq_{\varepsilon} F', \Psi$. First observe that $F', \Psi = F, \Psi[t/x]$. Since Ψ implies the equation $\exists y \,.\, x \simeq t$, every model of F implies there is a solution to y such that $\Psi[t/x]$ that agrees with the variables in F. Conversely, if $F, \Psi[t/x]$ is satisfied by $\mathcal{M}'$, then $\mathcal{M}'$ already satisfies F.

Lemma 7. *Given $F \parallel \theta\langle x \leftarrow t; \Psi\rangle^{\mathbb{B}}\theta' \overset{\text{Undo}}{\Longrightarrow} F, \Psi\theta' \parallel \theta\theta'$, s.t. $F_0 \succeq_{\theta\langle x \leftarrow t; \Psi\rangle^{\mathbb{B}}\theta'} F$, then $F_0 \succeq_{\theta\theta'} F, \Psi\theta'$ holds.*

Proof. Given an inference $F_1 \parallel \theta \Longrightarrow F_2 \parallel \theta\langle x \leftarrow t; \Psi\rangle^{\mathbb{B}}$. Lemma 6 establishes that the formula F_1 simplifies to F_2, Ψ modulo ϵ. Lemma 5 establishes that F_2, Ψ simplifies to $F, \Psi\theta'$ modulo θ'. Chaining the definition of simplification modulo transitively establishes the lemma.

With Corollary 4 and Lemma 7 we have then established Proposition 2.

It is worth examining why the side-conditions for simplification modulo are used. As the following example shows, transformations that only preserve satisfiability but strengthen formulas cannot be used easily in an incremental setting.

Example 6. Let F_0 be the satisfiable formula $x \simeq y \wedge y \leq z \wedge z \simeq v$. In that formula x, y are equal, and z, v are equal. Lets assume that we simplify via the solution where the classes are merged (i.e. where $y \simeq z$). It is satisfiability preserving. It suggests a transformation that we call Flex†.

$$\frac{x \simeq y \wedge y \leq z \wedge z \simeq v \parallel \varepsilon}{x \simeq z \wedge z \simeq v \parallel \langle y \leftarrow z; (x \simeq y \wedge y \leq z)\rangle^{\top}} \ \text{Flex}^{\dagger}$$

The resulting state is still satisfiable. Now Undo can be applied without any problems. The result is still satisfiable, but not equivalent to F_0 (does not have the models where the two equivalence classes are not merged).

$$\frac{x \simeq z \wedge z \simeq v \parallel \langle y \leftarrow z; (x \simeq y \wedge y \leq z)\rangle^{\top}}{(x \simeq y \wedge y \leq z) \wedge x \simeq z \wedge z \simeq v \parallel \varepsilon} \ \text{Undo}$$

Adding the constraint $y \simeq z - 1$ to F_0 would be satisfiable, but adding it to our formula is unsatisfiable.

4 Simplification Methods

Many simplification methods used in practice during pre-processing are equivalence preserving. These methods include formula rewriting, constant propagation, NNF conversion, quantifier elimination, and bit-blasting. They do not require the methodology from this paper and have been integral in Z3 since its inception. We will here discuss main simplification pre-processing routines that do not preserve equivalence and how they relate to our taxonomy.

4.1 Equality Solving

One of the most useful pre-processing techniques eliminates symbols when they can be *solved*, that is, a constraint implies an equality $x \simeq t$, where t is a term that does not contain x. Equality solving corresponds to finding unitary solutions to unification problems modulo theories. Most uses of equality solving are captured by transformations justified by rule Rigid. In Z3, equality solving comprises of a two stage process:

1. Extract a set of solution candidates $\mathcal{E}$ implied by the current formula φ.
2. Extract from $\mathcal{E}$ a subset of *solutions* that can be oriented without introducing cyclic dependencies.

To elaborate, let $\mathcal{E}$ be a set of solution candidates $x_1 = t_1, \ldots x_n = t_n$. The candidates may contain multiple equalities using the same symbol. For example, $\mathcal{E}$ could be $x = f(x), x = g(y), y = h(z)$. We can't use the solution $x = f(x)$ because x already occurs in $f(x)$. But we can use the solution $x = g(y), y = h(z)$ processed in this order as first x is replaced by $g(y)$, then y is replaced by $h(z)$. In the second stage we extract from $\mathcal{E}$ a subset of equalities $x_{i_1} = t_{i_1}, \ldots, x_{i_k} = t_{i_k}$, where x_{i_j} are distinct and t_{i_j} are terms such that $x_{i_j} \not\in t_{i_{j'}}$ for $j \leq j'$. The subset is in triangular form.

Example 7. We illustrate two application of Rigid for eliminating two symbols from three equations. The choice of the first two equations is arbitrary. An alternative simplification could choose to eliminate x and z instead. It is not possible, however, to eliminate all three variables.

$$
\begin{array}{l}
F, x \simeq y+1, y \simeq z+1, z \simeq f(x) \parallel \theta \Longrightarrow^{\textsc{Rigid}} \\
F[y+1/x], y \simeq z+1, z \simeq f(y+1) \parallel \theta\langle x \leftarrow y+1; x \simeq y+1\rangle^{\perp} \Longrightarrow^{\textsc{Rigid}} \\
F[y+1/x, z+1/y], z \simeq f(z+2) \parallel \theta\langle x \leftarrow y+1; x \simeq y+1\rangle^{\perp}\langle y \leftarrow z+1; y \simeq z+1\rangle^{\perp}
\end{array}
$$

The set of unification modulo theories facilities used in Z3 is based on extracting simple definitions. Foremost, for a conjunct $x \simeq t$ of φ, where x is uninterpreted, $x \neq t$, include the equality candidate $x \simeq t$. Other equality candidates are included from formulas of the form $ite(c, x \simeq t, x \simeq s)$ and arithmetic equalities of the form $x + s \simeq t$, such that $x \simeq t - s$ is a solution candidate for x. Note that solution candidates are not necessarily unique for an equality. The constraint $x + y \simeq t$ can be used as solution to both x and y. If x has a nested occurrence within t, the solution for y, but not x, can be used. Equality solving interacts with simplification pre-processing: equalities over algebraic data-types can be assumed to be in decomposed form already since rewriting simplification decomposes equalities of the form $\mathit{cons}(h_1, t_1) \simeq \mathit{cons}(h_2, t_2)$ into $h_1 \simeq h_2 \wedge t_1 \simeq t_2$. Equality solving can be extended modulo theories in several directions. Arithmetical equalities can be extracted from Diophantine equations solving and polynomial equality factorization as part of establishing a Gröbner basis. Equalities can be extracted from inequalities [6,31], other theories, such as the theory of arrays allow extracting solutions from equalities $\mathit{store}(a, i, v) \simeq t$, where a is a symbol that does not occur in t, i, v, as $a \simeq \mathit{store}(t, i, w)$, together with the constraint $\mathit{select}(t, i) \simeq v$, where w is fresh. We leave a study of the cost/benefits of these approaches within the context of incremental pre-processing to future work.

Equality solving is extended to sub-formulas in the following way: When a positive sub-formula implies an equality $x \simeq t$ and the symbol x does not occur outside of the sub-formula then x can be replaced by t within the subformula. The solution is no longer *rigid constrained* but can be justified by Flex.

Example 8. Suppose $x \notin F, \Psi$, then we can use Flex to justify the simplification

$$F, (x \simeq t \wedge \Phi[x]) \vee \Psi \parallel \theta \Longrightarrow^{\textsc{Flex}} F, \Phi[t] \vee \Psi \parallel \theta\langle x \leftarrow t; (x \simeq t \wedge \Phi[x]) \vee \Psi\rangle^{\top}$$

4.2 Unconstrained Sub-terms

Symbols that have a single occurrence in a formula may be solved for based on context. For example, with the formula $x \leq y, y < z, z \leq u, p(u), q(u)$, the constant x can be eliminated by using the solution $x \simeq y$. Then y can be eliminated by setting $y \simeq z - 1$, and finally $z \simeq u$.

Invertibility of unconstrained symbols (see e.g. [7,8]) in an incremental setting for bit-vectors was introduced in [19]. The method implements the following proof-rule, exemplified for the term $x + t$, containing the only occurrence of x.

$$\begin{array}{ll}\textsc{Invert}: & \\ F[x+t] \parallel \theta \Longrightarrow F[y] \parallel \theta\langle x \leftarrow y - t; y \simeq x + t\rangle^{\top} & \text{if } x \text{ occurs uniquely in } F \\ & y \text{ is fresh}\end{array}$$

To justify rule Invert in our setting, it suffices to check the condition from Lemma 6. Alternatively, we can use the generic rule Update when applying unconstrained simplifications. The rule Invert is more efficient than using Update because the latter requires adding back an entire conjunction Ψ where the invertible term $x + t$ occurs. Invertibility can also be used to justify elimination of nested definitions. For a definition $F \wedge ((x \simeq t \wedge \Phi[x]) \vee \Psi)$ (see Example 8), where $x \notin F, \Psi$ can first be rewritten as $F \wedge ((x \simeq t \wedge \Phi[t]) \vee \Psi)$. Then $x \simeq t$ is invertible because it contains the only occurrence of x. The new constraint is $F \wedge ((y \wedge \Phi[t]) \vee \Psi)$ where y is a fresh Boolean symbol.

Invertibility conditions are theory dependent. Figure 4 exemplifies main invertibility conditions for arithmetic[1].

$$
\begin{array}{l}
F[t - x] \;\|\; \theta \Longrightarrow^{\textsc{Invert}} F[y] \;\|\; \theta\langle x \leftarrow t - y; y \simeq t - x\rangle^{\top} \\
F[x \cdot x'] \;\|\; \theta \Longrightarrow^{\textsc{Invert}} F[y] \;\|\; \theta\langle x, x' \leftarrow y, 1; y \simeq x \cdot x'\rangle^{\top} \\
F[x \leq t] \;\|\; \theta \Longrightarrow^{\textsc{Invert}} F[y] \;\|\; \theta\langle x \leftarrow \mathit{ite}(y, t, t+1); y \simeq x \leq t\rangle^{\top} \\
F[t \leq x] \;\|\; \theta \Longrightarrow^{\textsc{Invert}} F[y] \;\|\; \theta\langle x \leftarrow \mathit{ite}(y, t, t-1); y \simeq t \leq x\rangle^{\top}
\end{array}
$$

Fig. 4. Invertibility rules for symbols x, x' that occur uniquely in F; y is fresh.

Z3 uses a heap ordered by occurrence counts to identify candidates for invertibility. It first processes all symbols with occurrence count 1. If it is possible to eliminate a symbol with occurrence count 1, the occurrence counts of sub-terms under the term that gets eliminated are decreased. The elimination process stops once the heap only contains symbols with occurrence counts above 1.

4.3 Symbol Elimination and Macros

SAT solvers use symbol elimination [15] to simplify clauses. The first-order version [11] remains timely in more recent works as well [28]. A predicate p can be eliminated if it occurs at most once in every clause either positively or negatively. Clauses that contain p are replaced by resolvents by applying binary resolution exhaustively, and then remove clauses containing p.

Example 9. We illustrate symbol elimination for the ground case with two clauses, and F such that $p \notin F$, as an instance of the Update rule.

$$
\begin{array}{l}
F, p(t) \vee \Phi, \neg p(s) \vee \Psi \;\|\; \theta \Longrightarrow^{\textsc{Update}} \\
F, s \not\simeq t \vee \Phi \vee \Psi \;\|\; \theta\langle p \leftarrow \lambda x \,.\, p(x) \vee (x \simeq t \wedge \neg\Phi); p(t) \vee \Phi, \neg p(s) \vee \Psi\rangle^{\top}
\end{array}
$$

The same elimination technique can also be applied to Horn clauses where p does not occur both in the head and body of any rule. A solution for the eliminated predicate is a conjunction of the upper bounds for p or a disjunction of lower bounds for p. It is generally a quantified formula. If the involved clauses admit quantifier free interpolants, the solution can also be computed using an interpolant from a solution to the reduced system [4]. Thus, the term t in a substitution $x \mapsto t$ may only be computed *after* an initial model is known.

There are many cases where symbols can be eliminated incrementally and justified by the Rigid rule:

- Macros $\forall x \,.\, f(x) \simeq t[x]$, $\forall x \,.\, f(x) + s \simeq t$ are handled as $\forall x \,.\, f(x) \simeq t - s$, assuming f is not free in s, t. Then replace occurrences $f(a)$ by $t[a]$, respectively $t[a] - s[a]$.

[1] A summary of rules used for other theories can be found online: https://microsoft.github.io/z3guide/docs/strategies/summary#tactic-elim-uncnstr.

- Quasi macros $\forall x, y \;.\; f(x, y, x + y) \simeq t[x, y]$, then replace $f(a, b, c)$ by $ite(c \simeq a + b, t[a, b], f'(a, b, c))$, assuming $f \notin t$.
- Conditional macros $\forall x \;.\; f(x) \simeq t[x] \vee C[x]$, then replace $f(a)$ by $ite(C[a], f'(a), t[a])$, where $f \notin t, C$.
- $(f(x) \simeq t) \equiv \psi$, where $f \notin t, \psi$. Then replace $f(a)$ by $ite(\psi, t, f'(a))$ and add the clause $\forall x \;.\; f'(x) \not\simeq t$.

Macro elimination can be extended to ordered structures and in combination of theories [42]. It has been integral to making quantified reasoning with bit-vectors [44] practical. We claim that first-order in-processing rules based on blocked clauses, asymmetric tautology elimination, covered clauses known from SAT [29] can also be captured by Update. We substantiate the claim with an example, but leave a comprehensive treatment for future work:

Example 10. Consider the clause $C := p(x) \vee q(x)$ and $F := \neg p(x) \vee p(f(x)) \vee r(x), \neg p(x) \vee p(f(x)) \vee p(g(x))$. The variable x is universally quantified. Then C can be rewritten to $p(x) \vee q(x) \vee p(f(x))$ without affecting satisfiability. The covered literal $p(f(x))$ was added to C as it occurs in every resolvent with $p(x)$. The model for p has to be fixed, however. The model update is a first-order lifting of the propositional case.

$$\begin{aligned}&F, p(x) \vee q(x) \;\|\; \theta \Longrightarrow^{\textsc{Update}}\\ &F, p(x) \vee q(x) \vee p(f(x)) \;\|\; \theta\langle p \leftarrow \lambda x \;.\; p(x) \vee p(f(x)); \forall x \;.\; p(x) \vee q(x)\rangle^{\top}\end{aligned}$$

To illustrate unification constraints in model updates, consider the clause $C := p(h(x)) \vee q(x)$ and $p' := \lambda x \;.\; p(x) \vee \exists y \;.\; x \simeq h(y) \wedge \neg q(y)$:

$$\begin{aligned}&F, p(h(x)) \vee q(x) \;\|\; \theta \Longrightarrow^{\textsc{Update}}\\ &F, p(h(x)) \vee q(x) \vee p(f(h(x))) \;\|\; \theta\langle p \leftarrow p'; \forall x \;.\; p(h(x)) \vee q(x)\rangle^{\top}\end{aligned}$$

5 Implementation

We have implemented incremental pre-processing as an integral component of a new SMT solver, part of Z3. It can be enabled by setting the option `sat.smt=true` from the command line. It includes simplification by equality solving, elimination of uninterpreted sub-terms and macro detection as described in Sect. 4[2]. The primary reason for supporting incremental pre-processing has been usability. GitHub issues pointing to performance cliffs when switching to incremental mode are recurrent. A distilled example where pre-processing can solve formulas is as follows:

Example 11. Consider the benchmark.

```
(set-option :unsat_core true) (set-option :sat.smt true)
(declare-const exp Int) (push)
(assert (! (= exp 1) :named assumption))
(assert (not (= 2 (^ 2 exp)))) (check-sat) (get-unsat-core)
```

[2] See https://microsoft.github.io/z3guide/docs/strategies/simplifiers for a summary of simplifiers.

The legacy solver of z3 cannot solve it because it only knows about constant folding when expanding the definition of exponentiation (the symbol `^`). With incremental propagation, the equality `(not (= 2 (^ 2 exp)))` simplifies to `false`.

Simplifiers interoperate with user scopes: SMT solvers support scoping using operations `push` and `pop`. All assertions made within a `push` are invalidated by a matching `pop`. To allow simplifiers to inter-operate with recursive function definitions they track symbols used in the bodies of recursive functions as *frozen*. Those symbols are excluded from solving. Similar to CaDiCaL's implementation for replaying clauses (see [18]), our implementation of Replay stores the domain of θ in a hash-set to bypass processing formulas that have no symbols in θ.

6 Related Work

6.1 Pre- and In-processing for SAT and QBF

Pre-processing for SAT has received significant attention with the milestone work in Satellite [15] and then using notions of blocked clauses [27] and solution reconstruction [25]. Pre-processing techniques for QBF are discussed for example in [3, 22]. The main pre-processing methods for propositional satisfiability solvers can be captured using our rule Update (see Example 4 for an instance of blocked clause elimination simplification). For the case where $\neg p \vee D$ is a blocked clause, the model update is the de-Morgan dual: removing $\neg p \vee D$ triggers the update $\mathcal{M}[p \mapsto (p \wedge D)^{\mathcal{M}}]$.

The work [18] introduces an inference system that also addresses *redundant* clauses and represents model updates using a notion of *witness labeled clauses*. The semantic content of the rules used for SAT are captured by Update. However, we elided tracking redundant clauses in this work. The case for SMT motivate specialized rules Rigid, Flex and Invert.

6.2 Pre-processing for SMT

Pre-processing simplification is integral in all main SMT solvers, including [2, 33]. Incremental pre-processing with special attention to bit-vectors was introduced in [19]. Transformations considered in this thesis can be represented by the Rigid and Invert rules. Z3 exposes pre-processing simplifications as tactics [13] and allows users to compose them to suit specific needs of applications.

Invertibility conditions are used in [34] to guide local search. This work considers also a candidate value of all symbols. For example, $F[x \cdot t]$ is invertible to $F[y]$ if t evaluates to 1.

6.3 Pre-processing for MIP

Pre-solving is terminology for pre-processing for mixed-integer linear programming solvers. There is a significant repertoire of pre-solving methods integrated in leading MIP solvers. Their effects are well documented in the newer survey

[1], which provides an updated perspective to [20]. Pre-solving was developed earlier in [40]. The main methods can be categorized as operating on single rows (single constraints) or single columns (single variables), multiple rows, and multiple columns, and use global information about the tableau. They include also methods known from other domains, such as literal probing also found in SAT solvers, and symmetry reduction for sparse systems [38]. We are not aware of under-constrained simplifications used in mainstream MIP solvers. Only symmetry reduction stands out as outside the scope of incremental pre-solve methods.

Example 12. Pre-processing that combines two rows or combines two columns relies on efficient indexing [21] to be effective. The two column non-zero cancellation method considers the situation where the coefficients to two variables maintain a high degree of correlation. Consider the following formula

$$2x + 4y + z \leq 5 \ \wedge \ x + 2y + u \leq 6 \ \wedge \ 3x + y + z \leq 3 \ \wedge \ \varphi \text{ where } x, y \notin \varphi.$$

The coefficients to x, y in the first two inequalities are related by the affine relation given by $\lambda = 2$. In this case the system can be reformulated, justified by rule Rigid, by introducing a fresh variable v and using the inequalities

$$2v + z \leq 5 \ \wedge \ v + u \leq 6 \ \wedge \ 3v - 5y + z \leq 3 \ \wedge \ \varphi.$$

6.4 Pre-processing in First- and Higher-Order Provers

Pre-processing is also an important part of first-order theorem provers. Techniques for creating small clausal normal forms have long attracted attention [35]. Main simplifications [24] are based on detecting definitions similar to what is described in Sect. 4.3, but with the extra twist of ensuring that simplifications preserve first-order decidability, such as ensuring that formulas remain within the EPR fragment. Furthermore a variant of AIGs with nodes representing quantifiers are used to detect shared structure. While [24] is only concerned establishing preservation of satisfiability we note that the classification as model equivalent from Sect. 4.3 extends to the cases considered. In-processing inspired by SAT was pursued for first-order [29,43] and recently for higher-order settings [5].

6.5 Constrained Horn Clauses

Constrained Horn Clauses [4], enjoy a tight connection with Logic Programming where several transformation techniques were developed [10,12], including incremental consequence propagation [36]. *Fold* [9] transformations introduce auxiliary predicates and rules that correspond to replacing a code-block with an auxiliary procedure. It is justified by Rigid. *Unfold* transformations can be justified by Update and correspond to macro elimination.

7 Summary

We introduced a calculus of pre-processing for SMT. It distinguishes simplifications that are *rigid* and so can be applied to new formulas as substitutions. Other simplified formulas may need to be re-introduced similar to re-introducing removed clauses in SAT. We examine several of the pre-processing methods studied in SAT, ATP, MIP and SMT as instances of the calculus. We leave empirical and algorithmic studies of new pre- and in-processing methods to future work. Another angle we have left on the table is reconciling pre-processing with in-processing. For SAT, it was useful to develop a calculus that accounted for both irredundant and redundant clauses. In our current effort we have set this angle aside in favour of establishing main properties on replaying substitutions.

Acknowledgment. Thanks to the reviewers for their extensive constructive feedback and to Diego Olivier Fernandez Pons for introducing us to MIP pre-solving. The research was partially funded by the Austrian Science Fund (FWF) under project No. T-1306.

References

1. Achterberg, T., Bixby, R.E., Zonghao, G., Rothberg, E., Weninger, D.: Presolve reductions in mixed integer programming. INFORMS J. Comput. **32**(2), 473–506 (2020)
2. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
3. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 101–115. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_10
4. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
5. Blanchette, J., Vukmirović, P.: SAT-inspired higher-order eliminations (2023)
6. Bromberger, M., Weidenbach, C.: New techniques for linear arithmetic: cubes and equalities. Formal Methods Syst. Des. **51**(3), 433–461 (2017). https://doi.org/10.1007/s10703-017-0278-7
7. Brummayer, R.: Efficient SMT solving for bit vectors and the extensional theory of arrays. Ph.D. thesis, Johannes Kepler University of Linz (2010)
8. Bruttomesso, R.: RTL Verification: From SAT to SMT(BV). Ph.D. thesis, University of Trento (2008)
9. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. J. ACM **24**(1), 44–67 (1977)
10. Calzavara, S., Grishchenko, I., Maffei, M.: HornDroid: practical and sound static analysis of Android applications by SMT solving. In: IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, 21–24 March 2016, pp. 47–62. IEEE (2016)

11. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960)
12. De Angelis, E., Fioravanti, F., Gallagher, J.P., Hermenegildo, M.V., Pettorossi, A., Proietti, M.: Analysis and transformation of constrained horn clauses for program verification. CoRR, abs/2108.00739 (2021)
13. de Moura, L., Passmore, G.O.: The strategy challenge in SMT solving. In: Bonacina, M.P., Stickel, M.E. (eds.) Automated Reasoning and Mathematics. LNCS (LNAI), vol. 7788, pp. 15–44. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36675-8_2
14. Déharbe, D., Fontaine, P., Merz, S., Woltzenlogel Paleo, B.: Exploiting symmetry in SMT problems. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 222–236. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_18
15. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_5
16. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
17. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press (1972)
18. Fazekas, K., Biere, A., Scholl, C.: Incremental inprocessing in SAT solving. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 136–154. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_9
19. Franzén, A.: Efficient solving of the satisfiability modulo bit-vectors problem and some extensions to SMT. Ph.D. thesis, University of Trento, Italy (2010)
20. Gamrath, G., Koch, T., Martin, A., Miltenberger, M., Weninger, D.: Progress in presolving for mixed integer programming. Math. Program. Comput. **7**(4), 367–398 (2015). https://doi.org/10.1007/s12532-015-0083-5
21. Gemander, P., Chen, W., Weninger, D., Gottwald, L., Gleixner, A.M., Martin, A.: Two-row and two-column mixed-integer presolve using hashing-based pairing methods. EURO J. Comput. Optim. **8**(3), 205–240 (2020)
22. Giunchiglia, E., Marin, P., Narizzano, M.: sQueezeBF: an effective preprocessor for QBFs based on equivalence reasoning. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 85–98. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14186-7_9
23. Heule, M.J.H., Kiesl, B., Biere, A.: Strong extension-free proof systems. J. Autom. Reason. **64**(3), 533–554 (2020)
24. Hoder, K., Khasidashvili, Z., Korovin, K., Voronkov, A.: Preprocessing techniques for first-order clausification. In: Cabodi, G., Singh, S. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, 22–25 October 2012, pp. 44–51. IEEE (2012)
25. Järvisalo, M., Biere, A.: Reconstructing solutions after blocked clause elimination. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 340–345. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14186-7_30
26. Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 129–144. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_10

27. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 355–370. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_28
28. Khasidashvili, Z., Korovin, K.: Predicate elimination for preprocessing in first-order theorem proving. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 361–372. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_22
29. Kiesl, B., Suda, M.: A unifying principle for clause elimination in first-order logic. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 274–290. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_17
30. Kiesl, B., Suda, M., Seidl, M., Tompits, H., Biere, A.: Blocked clauses in first-order logic. In: Eiter, T., Sands, D. (eds.) LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, 7–12 May 2017. EPiC Series in Computing, vol. 46, pp. 31–48. EasyChair (2017)
31. Kincaid, Z., Koh, N., Zhu, S.: When less is more: consequence-finding in a weak theory of arithmetic. Proc. ACM Program. Lang. **7**(POPL), 1275–1307 (2023)
32. Kullmann, O.: On a generalization of extended resolution. Discret. Appl. Math. **96–97**, 149–176 (1999)
33. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisf. Boolean Model. Comput. **9**(1), 53–58 (2014)
34. Niemetz, A., Preiner, M., Biere, A.: Precise and complete propagation based local search for satisfiability modulo theories. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 199–217. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_11
35. Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 2, pp. 335–367. Elsevier and MIT Press (2001)
36. Puebla, G., Hermenegildo, M.: Optimized algorithms for incremental analysis of logic programs. In: Cousot, R., Schmidt, D.A. (eds.) SAS 1996. LNCS, vol. 1145, pp. 270–284. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61739-6_47
37. Reeves, J.E., Heule, M.J.H., Bryant, R.E.: Preprocessing of propagation redundant clauses. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 106–124. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_8
38. Sakallah, K.A.: Symmetry and satisfiability. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, 2nd edn., vol. 336, pp. 509–570. IOS Press (2021)
39. Saouli, S., Baarir, S., Dutheillet, C., Devriendt, J.: CosySEL: improving SAT solving using local symmetries. In: Dragoi, C., Emmi, M., Wang, J. (eds.) VMCAI 2023. LNCS, vol. 13881, pp. 252–266. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-24950-1_12
40. Savelsbergh, M.W.P.: Preprocessing and probing techniques for mixed integer programming problems. INFORMS J. Comput. **6**(4), 445–454 (1994)
41. Schöning, U.: Logik für Informatiker. Reihe Informatik, vol. 56. Bibliographisches Institut (1987)
42. Sofronie-Stokkermans, V.: Hierarchical and modular reasoning in complex theories: the case of local theory extensions. In: Konev, B., Wolter, F. (eds.) FroCoS 2007. LNCS (LNAI), vol. 4720, pp. 47–71. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74621-8_3

43. Vukmirović, P., Blanchette, J., Heule, M.J.H.: SAT-inspired eliminations for superposition. ACM Trans. Comput. Log. **24**(1), 7:1–7:25 (2023)
44. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: Efficiently solving quantified bit-vector formulas. Formal Methods Syst. Des. **42**(1), 3–23 (2013)
45. Cunxi, Yu., Ciesielski, M., Mishchenko, A.: Fast algebraic rewriting based on and-inverter graphs. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(9), 1907–1911 (2018)

Verified Given Clause Procedures

Jasmin Blanchette[1,2,3](✉), Qi Qiu[4], and Sophie Tourret[2,3]

[1] Ludwig-Maximilians-Universität München, Munich, Germany
jasmin.blanchette@ifi.lmu.de
[2] Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
sophie.tourret@inria.fr
[3] Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany
[4] Université Claude Bernard Lyon-1, LIRIS CNRS UMR 5205, Université de Lyon, Lyon, France
qi.qiu@univ-lyon1.fr

Abstract. Resolution and superposition provers rely on the given clause procedure to saturate clause sets. Using Isabelle/HOL, we formally verify four variants of the procedure: the well-known Otter and DISCOUNT loops as well as the newer iProver and Zipperposition loops. For each of the variants, we show that the procedure guarantees saturation, given a fair data structure to store the formulas that wait to be selected. Our formalization of the Zipperposition loop clarifies some fine points previously misunderstood in the literature.

Keywords: Saturation provers · Proof assistants · Stepwise refinement

1 Introduction

Resolution [13] and superposition [2] provers are based on *saturation*. In a first approximation, these provers perform all possible inferences between the available clauses. The full truth, however, is more complex: Provers may delete clauses that are considered *redundant*; for example, with resolution, if $\mathsf{p}(x)$ is in the clause set, then both $\mathsf{p}(\mathsf{a})$ and $\mathsf{p}(x) \lor \mathsf{q}(x)$ are redundant and could be deleted.

The procedure that saturates a set of clauses—or more generally, formulas—up to redundancy is called the *given clause procedure* [10, Sect. 2.3]. It has several variants. The two main variants are the Otter loop [10] and the DISCOUNT loop [1]. In this paper, we also consider the iProver [8] and Zipperposition loops [17]; they are variants of the Otter and DISCOUNT loops, respectively.

In its simplest form, the procedure distinguishes between *passive* and *active* formulas. Formulas start as passive. One passive formula is selected as the *given clause.*[1] Deletions and simplifications with respect to other passive and active formulas are then performed; for example, if the given clause is redundant with

[1] We keep the traditional name "given clause" even though our formulas are not necessarily clauses.

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 61–77, 2023.
https://doi.org/10.1007/978-3-031-38499-8_4

respect to an active formula, the given clause can be deleted, and if the given clause makes an active formula redundant, that formula can be deleted. Moreover, simplifications can take place; for example, in a superposition prover, if the term order specifies $\mathsf{b} \succ \mathsf{a}$, the given clause is $\mathsf{b} \approx \mathsf{a}$, and $\mathsf{p}(\mathsf{b})$ is an active formula, the active formula can be simplified to $\mathsf{p}(\mathsf{a})$ and made passive again.

Next, if the given clause has not been deleted, it is moved to the active set. All inferences between the given clause and formulas in the active set are then performed, and the resulting conclusions are put in the passive set. This procedure is repeated, starting with the selection of a new given clause, until the distinguished formula $\bot$ is derived or the passive set is empty.

The main metatheorem about this procedure states that if the given clause is chosen fairly (i.e., no passive formula is ignored forever), then the active set will be saturated (up to redundancy) at the limit. As a corollary, if the proof calculus is refutationally complete (i.e., it derives $\bot$ from any inconsistent set), then the prover based on the calculus will be refutationally complete as well.

We present an Isabelle/HOL [12] formalization of the Otter, DISCOUNT, iProver, and Zipperposition loops, culminating in a statement and proof of the main metatheorem for each one. We build on the pen-and-paper *saturation framework* developed by Waldmann, Tourret, Robillard, and Blanchette [18,19] and formalized in Isabelle/HOL by Tourret and Blanchette [16]. The framework is an elaboration of Bachmair–Ganzinger-style saturation [3, Sect. 4]. Waldmann et al. include descriptions of the four "loops" as instances of the framework, as Examples 71, 74, 81, and 82 [19]; our formalization follows these descriptions.

Among the four loops, the oldest one is the Otter loop. It originates from Otter, a resolution-based theorem prover for first-order logic introduced in 1988 [11]. Otter was the first prover to present the given clause algorithm, in its simplest form as described above.

The DISCOUNT loop followed a few years later as a byproduct of the DISCOUNT system [7], itself built to distribute proof tasks among processors. What distinguishes a DISCOUNT loop is that it really treats the passive set as passive. Its formulas serve only as the pool from which to choose the next given clause; they are never involved in deletions or other simplifications. Another key difference between the two loops is the decoupling of the scheduling of an inference and the production of its conclusion, which makes DISCOUNT able to propagate deletions and simplifications to discard inferences before their conclusions enter the passive set. For example, suppose that, in DISCOUNT, an inference

$$\frac{\mathsf{p}(x) \vee \mathsf{p}(\mathsf{a}) \qquad \neg\, \mathsf{p}(y) \vee \mathsf{q}(y)}{\mathsf{p}(x) \vee \mathsf{q}(y)}$$

called ι is scheduled, in a derivation using first-order resolution. Then suppose that, before ι is realized, $\mathsf{p}(\mathsf{a})$ is generated (e.g., as the result of the factorization of $\mathsf{p}(x) \vee \mathsf{p}(\mathsf{a})$). This triggers the deletion of $\mathsf{p}(x) \vee \mathsf{p}(\mathsf{a})$, which has become redundant. Then ι becomes an orphan inference, since one of its premises is no longer in the active set. It can be deleted without threatening the procedure's completeness. In contrast, in an Otter loop, if ι is scheduled before $\mathsf{p}(\mathsf{a})$ is selected as the given clause, ι's conclusion $\mathsf{p}(x) \vee \mathsf{q}(y)$ is directly added to the passive set, where it can be simplified.

What we call the iProver loop [8] is an extension of the Otter loop with a transition that removes a formula C if C is made redundant by a formula set M. This terminology is from Waldmann et al. [19, Example 74]. This rule, introduced when iProver was extended to handle the superposition calculus [8], combines an inference step with a step that simplifies the active set.

The last and most elaborate loop variant we present is the Zipperposition loop. Zipperposition is a higher-order theorem prover based on λ-superposition [4]. Its given clause procedure is designed to work with higher-order logic. Due to the explosiveness of higher-order unification, a single pair of premises can yield infinitely many conclusions. For example, the higher-order resolution inference

$$\frac{\mathsf{p}\,(\mathsf{f}\,(y\,\mathsf{a})) \vee \mathsf{q}\,y \qquad \neg\,\mathsf{p}\,(z\,(\mathsf{f}\,\mathsf{a}))}{\mathsf{q}\,(\lambda x.\,\mathsf{f}\,(\ldots(\mathsf{f}\,x)\ldots))}$$

where y and z are variables, produces infinitely many conclusions of the form $\mathsf{q}\,(\lambda x.\mathsf{f}^n\,x)$ for $n \in \mathbb{N}$. Thus, the passive set must be able to store possibly infinite sequences of lazily performed inferences. The Zipperposition loop was described by Vukmirović et al. [17] and by Waldmann et al. [19, Example 82].[2] Vukmirović et al. describe the loop's implementation in Zipperposition, which we believe to be correct. In contrast, Waldmann et al. present an abstract version of the loop and connect it, via stepwise refinement, to their saturation framework, obtaining the main metatheorem. However, in the latter work, the details are not worked out. Thanks to the Isabelle formalization, we note and address several issues such that we now have a first rigorous—in fact, fully formal—presentation of the essence of the Zipperposition loop including the metatheorem.

Our work is part of IsaFoL (Isabelle Formalization of Logic), an effort that aims at developing a formal library of results about logic and automated reasoning [6]. The Isabelle files amount to about 7000 lines of code. They were developed using the 2022 edition of Isabelle and are available in the *Archive of Formal Proofs* (*AFP*) [5], where they are updated to follow Isabelle's evolution.

This work joins a long list of verifications of calculi and provers. We refer to Blanchette [6, Sect. 5] for an overview of such works. The most closely related works are the two proofs of completeness of Bachmair and Ganzinger's resolution prover RP, by Schlichtkrull, Blanchette, Traytel, and Waldmann [14] and by Tourret and Blanchette [16] as well as the proof of completeness of ordered (unfailing) completion by Hirokawa, Middeldorp, Sternagel, and Winkler [9]. Instead of focusing on a single prover, here we consider general prover architectures. Via refinement, our results can be applied to individual provers.

2 Abstract Given Clause Procedures

To prove the main metatheorem for each of the four loops, we build on the saturation framework. The framework defines two highly abstract given clause

[2] Both groups of researchers include Blanchette and Tourret.

procedures, called GC (“given clause”) and LGC (“lazy given clause”) [19, Sect. 4]. They are formalized in the file Given_Clause_Architectures.thy of the *AFP* entry `Saturation_Framework` [15].

GC is an idealized Otter-style loop. It operates on sets of labeled formulas. Formulas have the generic type $'f$, and labels have the generic type $'l$. One label, active, identifies active formulas, and the other labels correspond to passive formulas. GC is defined as a two-rule transition system $\leadsto_{\mathsf{GC}}$. In Isabelle syntax:

$$\begin{array}{l}
\textbf{inductive}\ (\leadsto_{\mathsf{GC}}) :: ('f \times 'l)\ \mathit{set} \Rightarrow ('f \times 'l)\ \mathit{set} \Rightarrow \mathit{bool}\ \textbf{where} \\
\quad \mathit{process}\colon\ N_1 = N \cup M \implies N_2 = N \cup M' \implies M \subseteq \mathsf{Red_F}\ (\mathsf{N} \cup M') \implies \\
\qquad \mathsf{active\_subset}\ M' = \emptyset \implies N_1 \leadsto_{\mathsf{GC}} N_2 \\
\ |\ \mathit{infer}\colon\ N_1 = N \cup \{(C, L)\} \implies N_2 = N \cup \{(C, \mathsf{active})\} \cup M \implies \\
\qquad L \neq \mathsf{active} \implies \mathsf{active\_subset}\ M = \emptyset \implies \\
\qquad \mathsf{Inf\_between}\ (\mathsf{fst}\ `\ \mathsf{active\_subset}\ N)\ \{C\} \\
\qquad\quad \subseteq \mathsf{Red_I}\ (\mathsf{fst}\ `\ (N \cup \{(C, \mathsf{active})\} \cup M)) \implies \\
\qquad N_1 \leadsto_{\mathsf{GC}} N_2
\end{array}$$

When presenting Isabelle code, we will focus on the main ideas and not explain all the Isabelle syntax or all the symbols that occur in the code. We refer to Waldmann et al. [19] for mathematical statements of the key concepts and to the Isabelle theory files for the formal definitions.

Informally, the transition relation $\leadsto_{\mathsf{GC}}$ is defined as an inductive predicate equipped with two introduction rules, *process* and *infer*. Both rules allow a transition from N_1 to N_2 under some conditions:

- The *process* rule replaces a subset M of N_1 by M'. This is possible only if the redundancy criterion ($\mathsf{Red_F}$) justifies the replacement and the formulas in M' are all made passive (i.e., the active subset of M' is the empty set). This rule models formula simplification and deletion, but also replacing a passive label by another, “greater” passive label.
- The *infer* rule makes a passive formula C active and performs all inferences between this formula and active formulas, yielding M. Strictly speaking, the inferences need not be performed at all; it suffices that M makes the inferences redundant.

The main metatheorem for GC states that if the set of passive formulas is empty at the limit of a derivation, the active formula set is saturated at the limit.

The lazy variant LGC generalizes the DISCOUNT loop. It operates on pairs (T, N), where $T :: {}'f\ \mathit{inference\ set}$ is a set of inferences that have been scheduled but not yet performed and $N :: ({}'f \times {}'l)\ \mathit{set}$ is a set of labeled formulas. It consists of four rules that can be summarized as follows:

- The *process* rule is essentially as in GC. It leaves the T component unchanged.
- The *schedule_ infer* rule makes a passive formula active and schedules all the inferences between this formula and active formulas by adding them to the T component.
- The *compute_ infer* rule actually performs a scheduled inference or otherwise ensures that it is made redundant by adding suitable formulas.

- The *delete_orphan_infers* rule can be used to delete a scheduled inference if one of its premises has been deleted.

The main metatheorem for LGC states that if the set of scheduled inferences and the set of passive formulas are empty at the limit of a derivation starting in an initial state, the active formula set is saturated at the limit.

3 Otter and iProver Loops

The Otter loop [10] works on five-tuples (N, X, P, Y, A), where N is the set of *new* formulas; X is a subsingleton (i.e., the empty set or a singleton $\{C\}$) storing a formula moving from N to P; P is the set of so-called *passive* formulas (although, strictly speaking, the formulas in N, X, and Y are also passive); Y is a subsingleton storing the *given clause*, which moves from P to A; and A is the set of *active* formulas. All the sets are finite in practice.

Initial states have the form $(N, \emptyset, \emptyset, \emptyset, \emptyset)$. Inferences are assumed to be finitary, meaning that the set of inferences with C and formulas from A as premises (formally written $\mathsf{Inf\_between}\ A\ \{C\}$) is finite if A is finite. Premiseless inferences are disallowed.

Otter Loop without Fairness. The first version of the Otter loop, formalized in Otter_Loop.thy, does not make any fairness assumption on the choice of the given clause. The guarantee it offers is correspondingly weak: If the sets N, X, P, and Y are empty at the limit of a derivation starting in an initial state, then A is saturated. But there is no guarantee that N, X, P, and Y are empty at the limit. Later in this section, we will show how to ensure this generically.

The transition system $\rightsquigarrow_{\mathsf{OL}}$ for the Otter loop without fairness is as follows:

$$
\begin{array}{l}
\textbf{inductive}\ (\rightsquigarrow_{\mathsf{OL}}) :: ({}'f \times \mathsf{OL\_label})\ \mathit{set} \Rightarrow ({}'f \times \mathsf{OL\_label})\ \mathit{set} \Rightarrow \mathit{bool}\ \textbf{where} \\
\quad \mathit{choose\_n}\colon\ C \notin N \implies \\
\qquad \mathsf{state}\ (N \cup \{C\},\ \emptyset,\ P,\ \emptyset,\ A) \rightsquigarrow_{\mathsf{OL}} \mathsf{state}\ (N,\ \{C\},\ P,\ \emptyset,\ A) \\
\mid\ \mathit{delete\_fwd}\colon\ C \in \mathsf{Red_F}\ (P \cup A) \lor (\exists C' \in P \cup A.\ C' \preceq\!\!\cdot\ C) \implies \\
\qquad \mathsf{state}\ (N,\ \{C\},\ P,\ \emptyset,\ A) \rightsquigarrow_{\mathsf{OL}} \mathsf{state}\ (N,\ \emptyset,\ P,\ \emptyset,\ A) \\
\mid\ \mathit{simplify\_fwd}\colon\ C \in \mathsf{Red_F}\ (P \cup A \cup \{C'\}) \implies \\
\qquad \mathsf{state}\ (N,\ \{C\},\ P,\ \emptyset,\ A) \rightsquigarrow_{\mathsf{OL}} \mathsf{state}\ (N,\ \{C'\},\ P,\ \emptyset,\ A) \\
\mid\ \mathit{delete\_bwd\_p}\colon\ C' \in \mathsf{Red_F}\ \{C\} \lor C \preceq\!\!\cdot\ C' \implies \\
\qquad \mathsf{state}\ (N,\ \{C\},\ P \cup \{C'\},\ \emptyset,\ A) \rightsquigarrow_{\mathsf{OL}} \mathsf{state}\ (N,\ \{C\},\ P,\ \emptyset,\ A) \\
\mid\ \mathit{simplify\_bwd\_p}\colon\ C' \in \mathsf{Red_F}\ C,\ C'' \implies \\
\qquad \mathsf{state}\ (N,\ \{C\},\ P \cup \{C'\},\ \emptyset,\ A) \rightsquigarrow_{\mathsf{OL}} \mathsf{state}\ (N \cup \{C''\},\ \{C\},\ P,\ \emptyset,\ A) \\
\mid\ \mathit{delete\_bwd\_a}\colon\ C' \in \mathsf{Red_F}\ \{C\} \lor C \preceq\!\!\cdot\ C' \implies \\
\qquad \mathsf{state}\ (N,\ \{C\},\ P,\ \emptyset,\ A \cup \{C'\}) \rightsquigarrow_{\mathsf{OL}} \mathsf{state}\ (N,\ \{C\},\ P,\ \emptyset,\ A) \\
\mid\ \mathit{simplify\_bwd\_a}\colon\ C' \in \mathsf{Red_F}\ (C,\ C'') \implies \\
\qquad \mathsf{state}\ (N,\ \{C\},\ P,\ \emptyset,\ A \cup \{C'\}) \rightsquigarrow_{\mathsf{OL}} \mathsf{state}\ (N \cup \{C''\},\ \{C\},\ P,\ \emptyset,\ A) \\
\mid\ \mathit{transfer}\colon\ \mathsf{state}\ (N,\ \{C\},\ P,\ \emptyset,\ A) \rightsquigarrow_{\mathsf{OL}} \mathsf{state}\ (N,\ \emptyset,\ P \cup \{C\},\ \emptyset,\ A) \\
\mid\ \mathit{choose\_p}\colon\ C \notin P \implies \\
\qquad \mathsf{state}\ (\emptyset,\ \emptyset,\ P \cup \{C\},\ \emptyset,\ A) \rightsquigarrow_{\mathsf{OL}} \mathsf{state}\ (\emptyset,\ \emptyset,\ P,\ \{C\},\ A)
\end{array}
$$

| *infer*: $\mathsf{Inf\_between}\ A\ \{C\} \subseteq \mathsf{Red_I}\ (A \cup \{C\} \cup M) \implies$
$\quad\mathsf{state}\ (\emptyset, \emptyset, P, \{C\}, A) \rightsquigarrow_{\mathsf{OL}} \mathsf{state}\ (M, \emptyset, P, \emptyset, A \cup \{C\})$

The state function converts a five-tuple into a set of labeled formulas—an equivalent representation that is often more convenient formally. The labels are New (for N), XX (for X), Passive (for P), YY (for Y), and Active (for A, corresponding to active in GC).

The first nine rules all refine GC's *process* rule, whereas the tenth rule, *infer*, refines GC's *infer*. More precisely: The first rule moves a formula from N to X. The second and third rules delete or simplify the formula in X. The fourth to seventh rules delete or simplify other formulas using the formula in X. The eight rule moves a formula from X to P. The ninth rule moves a formula from P to Y. And the tenth rule moves a formula from Y to A and performs all inferences with formulas in A or otherwise ensures that the inferences are made redundant.

Following Waldmann et al., the rules introducing new formulas—namely, the *simplify* rules and *infer*—allow adding arbitrary formulas to the state and are therefore not sound. Since the metatheorems are about completeness, there is no harm in allowing unsound steps, such as skolemization. If desired, soundness can be required simply by adding the assumption $N \models N'$ for each step $N \rightsquigarrow_{\mathsf{OL}} N'$ in a derivation.

Compared with most descriptions of the Otter loop in the literature, the above formalization (and Example 71 in Waldmann et al. [19] on which it is based) is abstract and nondeterministic, allowing arbitrary interleavings of deletions, simplifications, and inferences. By not commiting to a specific strategy, we keep our code widely applicable: Our abstract Otter loop can be used as the basis of refinement steps targetting a wide range of deterministic procedures implementing specific strategies. We will see the same approach used for all the loops. We note that Bachmair and Ganzinger made a similar choice for their ordered resolution prover RP [3, Sect. 4].

Otter Loop with Fairness. Below we introduce a fair version of the Otter loop, called $\rightsquigarrow_{\mathsf{OLf}}$ and formalized in Fair_Otter_Loop_Def.thy. This new version is closer to an implementation.

inductive $(\rightsquigarrow_{\mathsf{OLf}}) :: ('p, 'f)\ \mathit{OLf\_state} \Rightarrow ('p, 'f)\ \mathit{OLf\_state} \Rightarrow \mathit{bool}$ **where**
choose_n: $C \notin N \implies$
$\quad(N \cup \{C\}, \mathsf{None}, P, \mathsf{None}, A) \rightsquigarrow_{\mathsf{OLf}} (N, \mathsf{Some}\ C, P, \mathsf{None}, A)$
| *delete_fwd*: $C \in \mathsf{Red_F}\ (\mathsf{elems}\ P \cup A) \lor (\exists C' \in \mathsf{elems}\ P \cup A.\ C' \preceq\!\cdot\ C) \implies$
$\quad(N, \mathsf{Some}\ C, P, \mathsf{None}, A) \rightsquigarrow_{\mathsf{OLf}} (N, \mathsf{None}, P, \mathsf{None}, A)$
| *simplify_fwd*: $C' \prec_{\mathsf{S}} C \implies C \in \mathsf{Red_F}\ (\mathsf{elems}\ P \cup A \cup \{C'\}) \implies$
$\quad(N, \mathsf{Some}\ C, P, \mathsf{None}, A) \rightsquigarrow_{\mathsf{OLf}} (N, \mathsf{Some}\ C', P, \mathsf{None}, A)$

⋮

| *choose_p*: $P \neq \mathsf{empty} \implies$
$\quad(\emptyset, \mathsf{None}, P, \mathsf{None}, A) \rightsquigarrow_{\mathsf{OLf}}$
$\quad(\emptyset, \mathsf{None}, \mathsf{remove}\ (\mathsf{select}\ P)\ P, \mathsf{Some}\ (\mathsf{select}\ P), A)$
| *infer*: $\mathsf{Inf\_between}\ A\ \{C\} \subseteq \mathsf{Red_I}\ (A \cup \{C\} \cup M) \implies$
$\quad(\emptyset, \mathsf{None}, P, \mathsf{Some}\ C, A) \rightsquigarrow_{\mathsf{OLf}} (M, \mathsf{None}, P, \mathsf{None}, A \cup \{C\})$

The definition of $\leadsto_{\mathsf{OLf}}$ differs from that of $\leadsto_{\mathsf{OL}}$ in two main respects:

- The set P is organized as some form of queue, with operations such as select, which chooses the queue's next element; remove, which removes all occurrences of an element from the queue; and elems, which returns the set of the queue's elements. The queue is assumed to be *fair*, meaning that if select is called infinitely often, every element in the queue will eventually be chosen and the limit of P will be empty.
- Simplification (e.g., in *simplify_fwd*) is allowed only if the simplified formula C' is smaller than the original formula C according to some given well-founded order $\prec_{\mathsf{S}}$. In practice, simplifications are usually well founded, so this is not a severe restriction.

Also note that the state is now directly represented as a five-tuple (without the mediation of the state function), where the subsingletons are of type $'f\ option$, with values of the forms None and Some C.

Formula Queue. The queue that represents the passive formula set P is formalized in its own file, Prover_Queue.thy. The file defines an abstract type of queue and the operations on it (empty, select, add, remove, and elems). It also expresses the fairness assumption on the select function:

> If a sequence of queue operations starting from an empty queue contains infinitely many removals of the selected element, then the queue is empty at the limit.

Moreover, the file contains an example implementation of the queue as a FIFO queue. This ensures that the abstract requirements on the queue, including fairness, are satisfiable.

iProver Loop with Fairness. To obtain an iProver loop from an Otter loop, only one extra rule is needed. The fair version of the iProver loop is formalized in Fair_iProver_Loop.thy as follows:

$$\begin{array}{l}
\textbf{inductive}\ (\leadsto_{\mathsf{ILf}}) :: ('p, 'f)\ \mathit{OLf\_state} \Rightarrow ('p, 'f)\ \mathit{OLf\_state} \Rightarrow \mathit{bool}\ \textbf{where} \\
\quad \mathit{ol}\colon\ \mathit{St} \leadsto_{\mathsf{OLf}} \mathit{St}' \implies \mathit{St} \leadsto_{\mathsf{ILf}} \mathit{St}' \\
\mid\ \mathit{red\_by\_children}\colon\ C \in \mathsf{Red_F}\ (A \cup M) \vee M = \{C'\} \wedge C' \prec\!\!\cdot\ C \implies \\
\qquad (\emptyset, \mathsf{None}, P, \mathsf{Some}\ C, A) \leadsto_{\mathsf{ILf}} (M, \mathsf{None}, P, \mathsf{None}, A)
\end{array}$$

The first rule, *ol*, executes any $\leadsto_{\mathsf{OLf}}$ rule as an iProver loop rule. The second rule, *red_by_children*, replaces a formula C by a set of formulas M that make it redundant. As M, iProver would use a set of simplified formulas produced by inferences with C as a premise and formulas from $A \cup \{C\}$ as further premises. The rule is stated in a more general, unsound form.

We prove the main metatheorem first for the iProver loop. Then, since an Otter derivation is an iProver derivation (in which the second rule, *red_by_children*, is not used), the result carries over directly to the Otter loop. The Isabelle statement, located in Fair_iProver_Loop.thy, is as follows:

theorem *fair_IL_Liminf_saturated*
assumes
full_chain ($\leadsto_{\mathsf{ILf}}$) *Sts* **and**
is_initial_OLf_state (*Sts* ! 0)
shows saturated (Liminf *Sts*)

Informally, this states that if *Sts* is a complete $\leadsto_{\mathsf{ILf}}$ derivation starting in a state of the form (N, None, empty, None, $\emptyset$), then the limit is saturated. The limit (strictly speaking, limit inferior) is defined by

$$\mathsf{Liminf}\ Xs = \bigcup_{i<|Xs|} \bigcap_{j:i\le j \wedge j<|Xs|} Xs\ !\ j$$

where Xs ! j returns the element at index j of the finite or infinite sequence Xs. In Isabelle, such sequences are represented by the type *'a llist* of "lazy lists."

This metatheorem is proved within the scope of the passive set queue's fairness assumption. It is derived from the metatheorem about the transition system $\leadsto_{\mathsf{IL}}$ without fairness, which is inherited from the abstract procedure GC.

Proof Sketch. The main difficulty is to show that N, X, P, and Y are empty at the limit. Once this is shown, we can apply the main metatheorem for GC, which states that if there are no passive formulas at the limit, the active formula set is saturated.

Let $St_0 \leadsto_{\mathsf{IL}} St_1 \leadsto_{\mathsf{IL}} \cdots$ be a complete derivation, where $St_i = (N_i, X_i, P_i, Y_i, A_i)$. If the derivation is finite, it is easy to show that the final state, and hence the limit, must be of the form ($\emptyset$, None, empty, None, A), as desired.

Otherwise, for an infinite derivation, we assume in turn that the limit of N, X, P, or Y is nonempty and show that this leads to a contradiction. We start with N. Let i be an index such that $N_i \cap N_{i+1} \cap \cdots \neq \emptyset$, which exists by the definition of limit. This means that $N_i, N_{i+1}, \ldots$ are all nonempty. By invariance, we can show that $Y_i, Y_{i+1}, \ldots$ are all empty. Thus, if we have a transition from St_j to St_{j+1} for $j \geq i$, it cannot be *infer* (via *ol*) or *red_by_children*. It can be shown that for the remaining transition rules, we have $St_i \sqsupset_1 St_{i+1} \sqsupset_1 \cdots$, where $\sqsupset_1$ is the converse of the lexicographic combination $\sqsubset_1$ of three well-founded relations:

- the multiset extension $\prec\!\!\!\prec_{\mathsf{S}}$ of $\prec_{\mathsf{S}}$ on entire states—i.e., on unions $N \cup X \cup P \cup Y \cup A$;
- as a tiebreaker, $\prec\!\!\!\prec_{\mathsf{S}}$ on N components;
- as a further tiebreaker, $\prec\!\!\!\prec_{\mathsf{S}}$ on X components.

Since the lexicographic combination of well-founded relations is well founded, the chain $St_i \sqsupset_1 St_{i+1} \sqsupset_1 \cdots$ cannot be infinite, a contradiction.

Next, we consider the X component. If X is nonempty forever, the only possible transition rules are deletions and simplifications, and both make the entire state decrease with respect to $\prec\!\!\!\prec_{\mathsf{S}}$. Again, we get a contradiction.

Next, we consider the P component. The fairness assumption for the queue guarantees that P is empty at the limit, at the condition that the *choose_p* rule is executed infinitely often. Since P is assumed not to be empty at the limit, *choose_p* must be executed only finitely often. Let i be an index from which no *choose_p* step takes place. We then have $St_i \sqsupset_2 St_{i+1} \sqsupset_2 \cdots$, where $\sqsupset_2$ is the converse of the lexicographic combination $\sqsubset_2$ of two well-founded relations:

- $\prec\!\!\cdot_{\mathsf{S}}$ on Y components;
- as a tiebreaker, the relation $\sqsubset_1$ on entire states.

Again, we get a contradiction.

Finally, for Y, the only two transitions possible, *infer* and *red_by_children*, are to a state where Y is empty afterward, contradicting the hypothesis that Y is nonempty forever. □

4 DISCOUNT Loop

The DISCOUNT loop [1] works on four-tuples (T, P, Y, A), where T is the set of *scheduled* ("to do") inferences, P is the set of so-called *passive* formulas (although, strictly speaking, any formula in Y is also passive); Y is a subsingleton storing the *given clause*; and A is the set of *active* formulas. All the sets are finite.

Initial states have the form $(\emptyset, P, \emptyset, \emptyset)$. Inferences are assumed to be finitary. We disallow premiseless inferences. Waldmann et al. [19, Example 81] allow them and let the T component of initial sets consist of all of them. However, in their "reasonable strategy," they implicitly assume that T is finite, in which case premiseless inferences can be immediately performed and replaced by the resulting formulas inserted in P.

DISCOUNT Loop without Fairness. The first version of the DISCOUNT loop, formalized in DISCOUNT_Loop.thy, does not make any fairness assumption on the choice of the inference to compute or the given clause. There is no guarantee that T, P, and Y are empty at the limit, but if they are, then A is saturated at the limit. Here is an extract of the definition, omitting the *delete_bwd* and *simplify_fwd* rules:

```
**inductive** $(\leadsto_{\mathsf{DL}}) :: {}'f\ inference\ set \times ({}'f \times DL\_label)\ set \Rightarrow$
    ${}'f\ inference\ set \times ({}'f \times DL\_label)\ set \Rightarrow bool$
**where**
    $compute\_infer\colon\ \iota \in \mathsf{Red_I}\ (A \cup \{C\}) \implies$
        $\mathsf{state}\ (T \cup \iota,\ P,\ \emptyset,\ A) \leadsto_{\mathsf{DL}} \mathsf{state}\ (T,\ P,\ \{C\},\ A)$
  | $choose\_p\colon\ \mathsf{state}\ (T,\ P \cup \{C\},\ \emptyset,\ A) \leadsto_{\mathsf{DL}} \mathsf{state}\ (T,\ P,\ \{C\},\ A)$
  | $delete\_fwd\colon\ C \in \mathsf{Red_F}\ A \lor (\exists C' \in A.\ C' \preceq\!\!\cdot\ C) \implies$
        $\mathsf{state}\ (T,\ P,\ \{C\},\ A) \leadsto_{\mathsf{DL}} \mathsf{state}\ (T,\ P,\ \emptyset,\ A)$
    $\vdots$
  | $simplify\_bwd\colon\ C' \in \mathsf{Red_F}\ \{C, C''\} \implies$
        $\mathsf{state}\ (T,\ P,\ C,\ A \cup \{C'\}) \leadsto_{\mathsf{DL}} \mathsf{state}\ (T,\ P \cup \{C''\},\ \{C\},\ A)$
  | $schedule\_infer\colon\ T' = \mathsf{Inf\_between}\ A\ \{C\} \implies$
        $\mathsf{state}\ (T,\ P,\ \{C\},\ A) \leadsto_{\mathsf{DL}} \mathsf{state}\ (T \cup T',\ P,\ \emptyset,\ A \cup \{C\})$
  | $delete\_orphan\_infers\colon\ T' \cap \mathsf{Inf\_from}\ A = \emptyset \implies$
        $\mathsf{state}\ (T \cup T',\ P,\ Y,\ A) \leadsto_{\mathsf{DL}} \mathsf{state}\ (T,\ P,\ Y,\ A)$
```

The state function converts a four-tuple (T, P, Y, A) into a pair (T, N), where N is a set of labeled formulas. The labels are $\mathsf{Passive}$ (for P), YY (for Y), and Active (for A, corresponding to active in LGC). The rules *compute_infer*, *schedule_infer*, and *delete_orphan_infers* refine the LGC rules of the same names; the other rules refine *process*.

DISCOUNT Loop with Fairness. In the fair version of the DISCOUNT loop, formalized in Fair_DISCOUNT_Loop.thy, the scheduled inferences and the passive formulas are organized as a single queue. A state is then a triple (P, Y, A), where P is the single queue that merges T and P from the above DISCOUNT loop, and Y and A are as above. Elements of P have the forms $\mathsf{Passive\_Inference}\ \iota$ and $\mathsf{Passive\_Formula}\ C$. The select function of P is assumed to be fair: If select is called infinitely often, every element in the queue will eventually be chosen and the limit of P will be empty.

The definition of the transition system is as follows:

```
inductive $(\rightsquigarrow_{\mathsf{DLf}}) :: ('p, 'f)\ \mathit{DLf\_state} \Rightarrow ('p, 'f)\ \mathit{DLf\_state} \Rightarrow \mathit{bool}$ where
  $\mathit{compute\_infer}$: $P \neq \mathsf{empty} \implies \mathsf{select}\ P = \mathsf{Passive\_Inference}\ \iota \implies$
      $\iota \in \mathsf{Red_I}\ (A \cup C) \implies$
      $(P, \mathsf{None}, A) \rightsquigarrow_{\mathsf{DLf}} (\mathsf{remove}\ (\mathsf{select}\ P)\ P, \mathsf{Some}\ C, A)$
 | $\mathit{choose\_p}$: $P \neq \mathsf{empty} \implies \mathsf{select}\ P = \mathsf{Passive\_Formula}\ C \implies$
      $(P, \mathsf{None}, A) \rightsquigarrow_{\mathsf{DLf}} (\mathsf{remove}\ (\mathsf{select}\ P)\ P, \mathsf{Some}\ C, A)$
 | $\mathit{delete\_fwd}$: $C \in \mathsf{Red_F}\ A \lor (\exists C' \in A.\ C' \preceq\!\!\cdot\ C) \implies$
      $(P, \mathsf{Some}\ C, A) \rightsquigarrow_{\mathsf{DLf}} (P, \mathsf{None}, A)$
   $\vdots$
 | $\mathit{simplify\_bwd}$: $C' \notin A \implies C'' \prec_{\mathsf{S}} C' \implies C' \in \mathsf{Red_F}\ \{C, C''\} \implies$
      $(P, \mathsf{Some}\ C, A \cup \{C'\}) \rightsquigarrow_{\mathsf{DLf}} (\mathsf{add}\ (\mathsf{Passive\_Formula}\ C'')\ P, \mathsf{Some}\ C, A)$
 | $\mathit{schedule\_infer}$: $\mathsf{set}\ \iota s = \mathsf{Inf\_between}\ A\ \{C\} \implies$
      $(P, \mathsf{Some}\ C, A) \rightsquigarrow_{\mathsf{DLf}}$
      $(\mathsf{fold}\ (\mathsf{add} \circ \mathsf{Passive\_Inference})\ \iota s\ P, \mathsf{None}, A \cup \{C\})$
 | $\mathit{delete\_orphan\_infers}$: $\iota s \neq [] \implies \mathsf{set}\ \iota s \subseteq \mathsf{passive\_inferences\_of}\ P \implies$
      $\mathsf{set}\ \iota s \cap \mathsf{Inf\_from}\ A = \emptyset \implies$
      $(P, Y, A) \rightsquigarrow_{\mathsf{DLf}} (\mathsf{fold}\ (\mathsf{remove} \circ \mathsf{Passive\_Inference})\ \iota s\ P, Y, A)$
```

We note the following:

- Inferences are added to P by *schedule_infer*. An inference can be deleted by *delete_orphan_infers* if one of the premises has been removed since the inference was scheduled.
- The next element from P is chosen by *compute_infer* or *choose_p*, depending on whether it is of the form $\mathsf{Passive\_Inference}\ \iota$ or $\mathsf{Passive\_Formula}\ C$.
- Formulas are added to P by *simplify_bwd*.

As with the Otter and iProver loops, the most important result is saturation at the limit:

theorem *fair_DL_Liminf_saturated*
  assumes
    full_chain ($\leadsto_{\mathsf{DLf}}$) *Sts* **and**
    is_initial_DLf_state (*Sts* ! 0)
  shows saturated (labeled_formulas_of (Liminf_fstate *Sts*))

Proof Sketch. The proof amounts to showing that the sets P and Y are empty at the limit. This is easy to show for finite derivations, so we focus on infinite ones. We proceed by contradiction. For P, the fairness assumption for the select function of the queue guarantees that P is empty at the limit, at the condition that the *compute_infer* and *choose_p* rules are collectively executed infinitely often. Since P is assumed not to be empty at the limit, these two rules must be executed only finitely often. Let i be an index from which no *compute_infer* or *choose_p* step takes place. We then have $St_i \sqsupset St_{i+1} \sqsupset \cdots$, where $\sqsupset$ is the converse of the lexicographic combination $\sqsubset$ of two well-founded relations:

- $<$ on the cardinality of Y components (0 or 1);
- as a tiebreaker, the multiset extension $\prec\!\!\prec_{\mathsf{S}}$ of $\prec_{\mathsf{S}}$ on unions $P \cup Y \cup A$.

Since the lexicographic combination of well-founded relations is well founded, the chain $St_i \sqsupset St_{i+1} \sqsupset \cdots$ cannot be infinite, a contradiction.

Finally, we consider Y. If Y is nonempty forever, the only possible transitions make the entire state decrease with respect to $\sqsubset$. This yields a contradiction. □

5 Zipperposition Loop

The Zipperposition loop [17] as described by Waldmann et al. [19, Example 82] works on four-tuples (T, P, Y, A), where the components have the same roles as in the DISCOUNT loop: T is the *scheduled* set, P is the *passive* set, Y is the *given clause*, if any, and A is the *active* set. For technical reasons, we need to enrich the state with a ghost component D ("done"), of type *'f inference set*, resulting in a five-tuple (T, D, P, Y, A). All the sets are finite.

The hallmark of the Zipperposition loop is that it can handle infinitary inferences. We assume that Inf_between A $\{C\}$ is countable if A is finite. (This assumption is implicit in Waldmann et al.) To store the infinitely many conclusions of an inference, T contains possibly infinite sequences of inferences, instead of individual inferences. Premiseless inferences are also allowed. Initial states have the form $(T, P, \emptyset, \emptyset, \emptyset)$, where T contains all the premiseless inferences of the underlying proof calculus and only those.

The implementation in Zipperposition by Vukmirović et al. [17] deviates from Waldmann et al. in one important respect: Instead of sequences of inferences, Zipperposition works with sequences of *subsingletons* of inferences. The special value $\emptyset$ is returned when no progress is made in computing an inference, to give control back to the given clause procedure. In the setting of Waldmann et al., this special value can be replaced by a tautology (e.g., $\top$ or $\top \approx \top$), which the given clause procedure can delete as redundant.

Zipperposition Loop without Fairness. The first version of the Zipperposition loop, formalized in Zipperposition_Loop.thy, does not make any fairness assumption on the choice of the inference to compute or the given clause. Here is an extract of the definition:

inductive ($\leadsto_{\mathsf{ZL}}$) :: $'f$ *inference set* $\times$ ($'f \times$ DL_label) *set* $\Rightarrow$
  $'f$ *inference set* $\times$ ($'f \times$ DL_label) *set* $\Rightarrow$ *bool*
where
  compute_infer: $\iota_0 \in \mathsf{Red_I}\ (A \cup \{C\}) \Longrightarrow$
    zl_state $(T + \{\mathsf{LCons}\ \iota_0\ \iota s\},\ D,\ P,\ \emptyset,\ A) \leadsto_{\mathsf{ZL}}$
    zl_state $(T + \{\iota s\},\ D \cup \{\iota_0\},\ P \cup \{C\},\ \emptyset,\ A)$
 | *choose_p*: zl_state $(T,\ D,\ P \cup \{C\},\ \emptyset,\ A) \leadsto_{\mathsf{ZL}}$ zl_state $(T,\ D,\ P,\ \{C\},\ A)$
 | *delete_fwd*: $C \in \mathsf{Red_F}\ A \lor (\exists C' \in A.\ C' \preceq\!\cdot\ C) \Longrightarrow$
    zl_state $(T,\ D,\ P,\ C,\ A) \leadsto_{\mathsf{ZL}}$ zl_state $(T,\ D,\ P,\ \emptyset,\ A)$
  $\vdots$
 | *schedule_infer*: inferences_of $T' =$ Inf_between $A\ \{C\} \Longrightarrow$
    zl_state $(T,\ D,\ P,\ C,\ A) \leadsto_{\mathsf{ZL}}$
    zl_state $(T + T',\ D -$ inferences_of $T',\ P,\ \emptyset,\ A \cup \{C\})$
 | *delete_orphan_infers*: set $\iota s \cap$ Inf_from $A = \emptyset \Longrightarrow$
    zl_state $(T + \{\iota s\},\ D,\ P,\ Y,\ A) \leadsto_{\mathsf{ZL}}$ zl_state $(T,\ D \cup$ set $\iota s,\ P,\ Y,\ A)$

The zl_state function converts a five-tuple (T, D, P, Y, A) into a pair (U, N), where

- U consists of all the inferences contained in T minus those in D (formally written inferences_of $T - D$); and
- N is a set of labeled formulas corresponding to P, Y, and A.

We use a multiset for the T component. Waldmann et al. use a set, but this is not very realistic because an implementation cannot in general detect duplicate infinite sequences.

The D component addresses a subtle issue in Waldmann et al. If we did not subtract D in the definition of U, the completeness theorem we would obtain from the LGC layer above would require the T component to be empty at the limit. However, a given inference ι might appear in T multiple times and hence always be present, even if we keep on removing copies of it, if new copies are continuously added. The issue goes away if we add ι to D whenever we compute it, in *compute_infer*—then the inference is not present in U (i.e., inferences_of $T - D$). In other words, computing an inference makes it momentarily disappear, even if there are multiple copies of it in T.

Admittedly, it is not easy to develop a robust intuitive understanding of how D works, but what matters ultimately is that D allows us to obtain a usable main metatheorem. The metatheorem states that if the set of scheduled inferences and the set of passive formulas are empty at the limit of a derivation starting in an initial state, the active formula set is saturated at the limit. We will also see, via an additional refinement layer, that the ghost component is truly a ghost and can be omitted once it has served its purpose.

Zipperposition Loop with Fairness. Unlike the fair DISCOUNT loop, the fair Zipperposition loop, formalized in Fair_Zipperposition_Loop.thy, keeps T and P separate. An extract of the Isabelle definition follows:

```
inductive ($\leadsto_{\mathsf{ZLf}}$) :: $('t, 'p, 'f)\ \mathit{ZLf\_state} \Rightarrow ('t, 'p, 'f)\ \mathit{ZLf\_state} \Rightarrow \mathit{bool}$
where
  compute_infer: $(\exists \iota s \in \mathsf{t\_llists}\ T.\ \iota s \neq \mathsf{LNil}) \Longrightarrow$
    $\mathsf{t\_pick\_elem}\ T = (\iota_0, T') \Longrightarrow \iota_0 \in \mathsf{Red_I}\ (A \cup \{C\}) \Longrightarrow$
    $(T,\ D,\ P,\ \mathsf{None},\ A) \leadsto_{\mathsf{ZLf}} (T',\ D \cup \{\iota_0\},\ \mathsf{p\_add}\ C\ P,\ \mathsf{None},\ A)$
| choose_p: $P \neq \mathsf{p\_empty} \Longrightarrow$
    $(T,\ D,\ P,\ \mathsf{None},\ A) \leadsto_{\mathsf{ZLf}}$
    $(T,\ D,\ \mathsf{p\_remove}\ (\mathsf{p\_select}\ P)\ P,\ \mathsf{Some}\ (\mathsf{p\_select}\ P),\ A)$
| delete_fwd: $C \in \mathsf{Red_F}\ A \vee (\exists C' \in A.\ C' \preceq\!\!\cdot\ C) \Longrightarrow$
    $(T,\ D,\ P,\ \mathsf{Some}\ C,\ A) \leadsto_{\mathsf{ZLf}} (T,\ D,\ P,\ \mathsf{None},\ A)$

  ⋮
| schedule_infer: $\mathsf{inferences\_of}\ \iota ss = \mathsf{Inf\_between}\ A\ \{C\} \Longrightarrow$
    $(T,\ D,\ P,\ \mathsf{Some}\ C,\ A) \leadsto_{\mathsf{ZLf}}$
    $(\mathsf{fold}\ \mathsf{t\_add\_llist}\ \iota ss\ T,\ \ D - \mathsf{inferences\_of}\ \iota ss,\ P,\ \mathsf{None},\ A \cup \{C\})$
| delete_orphan_infers: $\iota s \in \mathsf{t\_llists}\ T \Longrightarrow \mathsf{set}\ \iota s \cap \mathsf{Inf\_from}\ A = \emptyset \Longrightarrow$
    $(T,\ D,\ P,\ Y,\ A) \leadsto_{\mathsf{ZLf}} (\mathsf{t\_remove\_llist}\ \iota s\ T,\ D \cup \mathsf{set}\ \iota s,\ P,\ Y,\ A)$
```

The presence of two queues introduces some complications. Waldmann et al. [19, Example 82] claim that “to produce fair derivations, a prover needs to choose the sequence in ComputeInfer fairly and to choose the formula in ChooseP fairly.” However, this does not suffice: A counterexample would apply *compute_infer* infinitely often in a fair fashion, retrieving elements from some infinite sequences, without ever applying *choose_p* (whose choice of formula would then be vacuously fair). The solution is to add a fairness assumption stating that *compute_infer* is applied at most finitely many times before *choose_p* is applied—or, in other words, that if *compute_infer* is applied infinitely often, then so is *choose_p*. This leads to the following main metatheorem:

```
theorem fair_ZL_Liminf_saturated:
  assumes
    full_chain ($\leadsto_{\mathsf{ZLf}}$) $\mathit{Sts}$ and
    is_initial_ZLf_state ($\mathit{Sts}$ ! 0) and
    infinitely_often compute_infer_step $\mathit{Sts}$ $\longrightarrow$
      infinitely_often choose_p_step $\mathit{Sts}$
  shows saturated (labeled_formulas_of (Liminf_zl_fstate $\mathit{Sts}$))
```

Proof Sketch. Recall that zl_state maps (T, D, P, Y, A) to a pair (U, N). In the abstract LGC layer, U and the passive subset of N are required to be empty at the limit. To obtain the same effect in $\leadsto_{\mathsf{ZLf}}$, we must show that the sets U, P, and Y are empty at the limit. This is easy to show for finite derivations, so we focus on infinite ones. We proceed by contradiction.

We start with U. We first show that there must be infinitely many *compute_ infer* steps. Assume that there are finitely many. Then there exists an index i from which no more *compute_ infer* steps take place. We then have $St_i \sqsupset St_{i+1} \sqsupset \cdots$, where $\sqsupset$ is the converse of the lexicographic combination $\sqsubset$ of four well-founded relations:

- the multiset extension $\prec\!\!\prec_{\mathsf{S}}$ of $\prec_{\mathsf{S}}$ on unions $P \cup Y \cup A$;
- as a tiebreaker, $\prec\!\!\prec_{\mathsf{S}}$ on P components;
- as a further tiebreaker, $\prec\!\!\prec_{\mathsf{S}}$ on Y components;
- as a further tiebreaker, $<$ on the cardinality of T components.

We get a contradiction. Having shown that there are infinitely many *compute_ infer* steps, we exploit the queue's fairness to show that one of these steps will choose any given inference ι from the queue. Thanks to the D trick, ι will then momentarily vanish from U, ensuring that it is not in the limit. The same argument applies for any inference ι, showing that U is empty at the limit.

Next, we show that P is empty at the limit. We start by showing that there must be infinitely many *choose_ p* steps. Assume that there are finitely many. Then, by the third assumption, there must be finitely many *compute_ infer* steps as well. Let i be an index from which no more *compute_ infer* steps take place. We then have $St_i \sqsupset St_{i+1} \sqsupset \cdots$, as above, yielding a contradiction.

Finally, we show that Y is empty at the limit. Let i be an index such that $Y_i \cap Y_{i+1} \cap \cdots \neq \emptyset$. Since a *compute_ infer* step is possible only if Y is empty, no such steps are possible from index i. Again, we have $St_i \sqsupset St_{i+1} \sqsupset \cdots$, a contradiction. □

Queue of Formula Sequences. The queue data structure used for the T component of the Zipperposition loop needs to store a finite number of possibly infinite sequences of inferences. It is formalized in Prover_Lazy_List_Queue.thy. It provides the following operations on abstract queue and element types $'q$ and $'e$:

fixes
  empty :: $'q$ **and**
  add_ llist :: $'e\ \mathit{llist} \Rightarrow {'q} \Rightarrow {'q}$ **and**
  remove_ llist :: $'e\ \mathit{llist} \Rightarrow {'q} \Rightarrow {'q}$ **and**
  pick_ elem :: $'q \Rightarrow {'e} \times {'q}$ **and**
  llists :: $'q \Rightarrow {'e}\ \mathit{llist\ multiset}$

The fairness requirement on implementations of the abstract queue interface takes the following form:

> If a sequence of queue operations contains infinitely many pick_ elem steps and ι is at the head of one of the sequences stored in the queue, then either the sequence will be entirely removed (by orphan deletion) or ι will eventually be chosen.

A syntactically stronger formulation of fairness, where ι may occur anywhere in a sequence, is derived as a corollary:

> If a sequence of queue operations contains infinitely many pick_elem steps and ι occurs in one of the sequences stored in the queue at some index in the sequence, then either the sequence (possibly amputated from its leading elements) will be entirely removed or ι will eventually be chosen.

As a proof of concept, the theory file contains an example implementation of the queue as a FIFO queue. The proof that this FIFO queue is fair is the most finicky proof of our entire development.

Zipperposition Loop without Ghost Fields. In the last step of our development, we remove the D state component. D is useful to retrieve a usable main metatheorem for $\rightsquigarrow_{\mathsf{ZL}}$, but it is not explicitly referenced in the metatheorem for the fair variant $\rightsquigarrow_{\mathsf{ZLf}}$. The resulting transition system $\rightsquigarrow_{\mathsf{ZLfw}}$, formalized in Fair_Zipperposition_Loop_without_Ghosts.thy, operates on fourtuples (T, P, Y, A). Each transition is identical to the corresponding $\rightsquigarrow_{\mathsf{ZLf}}$ transition, omitting the D component. The main metatheorem is also essentially the same.

6 Conclusion

We presented an Isabelle/HOL formalization of four variants of the given clause procedure, starting from Tourret and Blanchette's formalization of two abstract given clause procedures [16]. We relied extensively on stepwise refinement to derive properties of more concrete transition systems from more abstract ones.

Our main findings concern the Zipperposition loop. We found that the refinement proof is not as straightforward as previously thought [19, Example 82] and requires a nontrivial abstraction function. In addition, we discovered a fairness condition—the necessity of avoiding computing inferences forever without selecting a formula—that was not mentioned before in the literature, and we clarified other fine points.

Acknowledgments. We thank Andrei Popescu for helping us with a coinductive proof that arose when removing the ghosts in the Zipperposition loop. We thank Uwe Waldmann for sharing with us his encyclopedic knowledge of the various given clause loops and their history. Finally, we thank Mark Summerfield and the anonymous reviewers for many helpful suggestions.

This research has received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

References

1. Avenhaus, J., Denzinger, J., Fuchs, M.: DISCOUNT: a system for distributed equational deduction. In: Hsiang, J. (ed.) RTA 1995. LNCS, vol. 914, pp. 397–402. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59200-8_72

2. Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. J. Log. Comput. **4**(3), 217–247 (1994). https://doi.org/10.1093/logcom/4.3.217
3. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, pp. 19–99. Elsevier and MIT Press (2001). https://doi.org/10.1016/b978-044450813-3/50004-7
4. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P.: Superposition for full higher-order logic. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 396–412. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_23
5. Blanchette, J., Qiu, Q., Tourret, S.: Given clause loops. Archive of Formal Proofs 2023 (2023). https://www.isa-afp.org/entries/Given_Clause_Loops.html
6. Blanchette, J.C.: Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In: Mahboubi, A., Myreen, M.O. (eds.) CPP 2019, pp. 1–13. ACM (2019). https://doi.org/10.1145/3293880.3294087
7. Denzinger, J., Pitz, W.: Das DISCOUNT-System: Benutzerhandbuch. SEKI working paper, Fachbereich Informatik, Univ. Kaiserslautern (1992). https://books.google.fr/books?id=8XwBvwEACAAJ
8. Duarte, A., Korovin, K.: Implementing superposition in iProver (system description). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12167, pp. 388–397. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_24
9. Hirokawa, N., Middeldorp, A., Sternagel, C., Winkler, S.: Infinite runs in abstract completion. In: Miller, D. (ed.) FSCD 2017. LIPIcs, vol. 84, pp. 19:1–19:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.FSCD.2017.19
10. McCune, W., Wos, L.: Otter–the CADE-13 competition incarnations. J. Autom. Reason. **18**(2), 211–220 (1997). https://doi.org/10.1023/A:1005843632307
11. McCune, W.W.: OTTER 3.0 reference manual and guide (1994). https://doi.org/10.2172/10129052, https://www.osti.gov/biblio/10129052
12. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
13. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965). https://doi.org/10.1145/321250.321253
14. Schlichtkrull, A., Blanchette, J., Traytel, D., Waldmann, U.: Formalizing Bachmair and Ganzinger's Ordered Resolution Prover. J. Autom. Reason. **64**(7), 1169–1195 (2020). https://doi.org/10.1007/s10817-020-09561-0
15. Tourret, S.: A comprehensive framework for saturation theorem proving. Archive of Formal Proofs 2020 (2020). https://www.isa-afp.org/entries/Saturation_Framework.html
16. Tourret, S., Blanchette, J.: A modular isabelle framework for verifying saturation provers. In: Hritcu, C., Popescu, A. (eds.) CPP 2021, pp. 224–237. ACM (2021). https://doi.org/10.1145/3437992.3439912
17. Vukmirović, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.: Making higher-order superposition work. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 415–432. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_24
18. Waldmann, U., Tourret, S., Robillard, S., Blanchette, J.: A comprehensive framework for saturation theorem proving. In: Peltier, N., Sofronie-Stokkermans, V.

(eds.) IJCAR 2020. LNCS (LNAI), vol. 12166, pp. 316–334. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_18

19. Waldmann, U., Tourret, S., Robillard, S., Blanchette, J.: A comprehensive framework for saturation theorem proving. J. Autom. Reason. **66**(4), 499–539 (2022). https://doi.org/10.1007/s10817-022-09621-7

QSMA: A New Algorithm for Quantified Satisfiability Modulo Theory and Assignment

Maria Paola Bonacina[1](✉), Stéphane Graham-Lengrand[2], and Christophe Vauthier[3]

[1] Università degli Studi di Verona, Verona, Italy
mariapaola.bonacina@univr.it
[2] SRI International, Menlo Park, USA
stephane.graham-lengrand@csl.sri.com
[3] École Normale Supérieure, Paris, France
cvauthier@clipper.ens.psl.eu

Abstract. This paper presents and proves totally correct a new algorithm, called QSMA, for the satisfiability of a quantified formula modulo a complete theory and an initial assignment. The optimized variant of QSMA implemented in YicesQS is described and shown to preserve total correctness. A report on the performance of YicesQS at the 2022 SMT competition is included. YicesQS ran in the LIA, NIA, LRA, NRA, and BV categories and ranked second for the "largest contribution" award (single queries). It was the only solver to solve all LRA instances, where it was about two orders of magnitude faster than the second best solver (Z3).

1 Introduction

Applications of automated reasoning generate formulas involving both quantifiers and symbols defined in background theories. For example, software verification needs reasoners that decide the satisfiability of quantified formulas modulo theories such as data structures and arithmetic (e.g., [20]). Therefore, endowing SMT solvers with quantifier reasoning (e.g., [3,9,11–14,22]), enriching first-order theorem provers with built-in theories (e.g., [1,2,19]), and integrating provers and solvers [7], are major research objectives.

If there is a single background theory $\mathcal{T}$, the $\mathcal{T}$-satisfiability of quantified formulas can be reduced to that of quantifier-free formulas if $\mathcal{T}$ admits quantifier elimination (QE): for every formula φ there exists a quantifier-free formula F that is $\mathcal{T}$-equivalent to φ. Since computing F can be prohibitively expensive (e.g., exponential in linear rational arithmetic (LRA) and doubly exponential in linear integer arithmetic (LIA) [8]), QE is not a practical solution.

In this paper we propose a practical solution in the form of a new algorithm called QSMA. In QSMA the computation of quantifier-free *model-based under-approximations* (MBU) and *model-based over-approximations* (MBO) of quantified formulas embodies a lazy approach to QE, which is tailored for

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 78–95, 2023.
https://doi.org/10.1007/978-3-031-38499-8_5

$\mathcal{T}$-satisfiability. MBU generates a quantifier-free implicant of the given formula that is true in the given model. *Model(-guided) generalization* for linear [12] and nonlinear real arithmetic (NRA) [17] is an instance of MBU. MBO generates a quantifier-free implied formula that is false in the given model. *Model interpolation* for NRA [17] is an instance of MBO.

The QSMA algorithm assumes that the theory $\mathcal{T}$ is *complete*. By its recursive nature, QSMA solves a generalized form of the satisfiability problem, called *quantified SMA* (*satisfiability modulo theory and assignment*): given a formula φ with *arbitrary quantification*, and an *initial assignment* to Boolean or first-order subterms of φ, find a theory model of φ that extends the initial assignment, or report that none exists. In addition to QSMA and its total correctness, we present an optimized variant named OptiQSMA, which preserves total correctness and is implemented in the YicesQS solver built on top of Yices 2. A report on experimental results from the 2022 SMT competition and a discussion complete the paper. We begin with a high-level view of QSMA.

1.1 High-Level View of the QSMA Algorithm

The QSMA algorithm works by progressively instantiating quantified variables. Consider a formula φ of the form $\exists \bar{x}_1.\forall \bar{x}_2.\exists \bar{x}_3 \ldots F[\bar{x}_1, \bar{x}_2, \bar{x}_3, \ldots]$ where F is quantifier-free. For example, suppose the theory is LRA, $\varphi = \exists x.\forall y.\exists z.F$ and $F = z \geq 0 \wedge x \geq 0 \wedge y + z \geq 0$. Say that QSMA assigns $x \leftarrow 0$. Whatever value is chosen for y, the algorithm can show that φ is true in LRA by assigning $z \leftarrow max(0, -y)$. If $F = z \geq 0 \wedge x \geq 0 \wedge y + z \leq 0$, no matter which (non-negative) value QSMA chooses for x, it can show that φ is false in LRA by picking $y \leftarrow 1$, because there is no value for z that satisfies $z \geq 0 \wedge z \leq -1$.

For an example that is not in prenex normal form, consider a formula φ of the form $\exists x.((\forall y_1.F_1[x, y_1]) \Rightarrow (\forall y_2.F_2[x, y_2]))$, where F_1 and F_2 are quantifier-free. QSMA sees the formula as $\exists x.((\exists y_1.\neg F_1[x, y_1]) \vee (\neg \exists y_2.\neg F_2[x, y_2]))$, and then as $\exists x.(p_1 \vee \neg p_2)$, where p_1 and p_2 are proxy Boolean variables for the quantified subformulas. QSMA assigns values to x, p_1, and p_2. If p_1 is assigned true, the algorithm tries to extend the assignment with a value for y_1 that satisfies $\neg F_1[x, y_1]$. If p_2 is assigned false, the algorithm tries to show that there is no value for y_2 that satisfies $\neg F_2[x, y_2]$.

Without loss of generality ($\neg\neg$ converts $\forall$ into $\neg\exists\neg$), we consider formulas

$$\varphi = \exists \bar{x}.F[\bar{z}, \bar{x}, \bar{p}]\{p_i \leftarrow \exists \bar{y}_i.G_i[\bar{z}, \bar{x}, \bar{y}_i]\}_{i=1}^{k}.$$

$F[\bar{z}, \bar{x}, \bar{p}]$ denotes a quantifier-free formula where the variables $\bar{z}$, $\bar{x}$, and $\bar{p}$ occur. Tuples $\bar{z}$ and $\bar{x}$ contain the first-order variables occurring free in F. Formula F is quantifier-free because the quantified subformulas $\varphi_i = \exists \bar{y}_i.G_i[\bar{z}, \bar{x}, \bar{y}_i]$ are replaced by proxy Boolean variables $\bar{p} = p_1, \ldots p_k$. Given an initial assignment to the free variables $\bar{z}$, we construct a QSMA-*tree* for φ. QSMA starts trying to satisfy $F[\bar{z}, \bar{x}, \bar{p}]$. If it fails, it means that φ is false under the initial assignment. If it succeeds, there are two cases. If $k = 0$, formula φ is true under the initial assignment. If $k > 0$, the algorithm descends recursively to consider the QSMA-subtrees for the φ_i subformulas ($1 \leq i \leq k$). If QSMA assigned true to p_i, it tries to show that φ_i is true. If QSMA assigned false to p_i, it tries to show that φ_i is false. If it succeeds for all QSMA-subtrees, formula φ is true under the initial assignment. For this, the model built by QSMA should satisfy $F[\bar{z}, \bar{x}, \bar{p}] \wedge \bigwedge_{i=1}^{n}(p_i \Leftrightarrow \varphi_i)$. Otherwise, formula φ is false under the initial assignment.

2 Preliminaries

A signature Σ is given by a set S of sorts and a set of sorted symbols. Given a class $\mathcal{V} = (\mathcal{V}^s)_{s \in S}$ of disjoint sets of sorted variables, $\Sigma[\mathcal{V}]$-formulas, Σ-sentences, and $\Sigma[\mathcal{V}]$-interpretations are defined as usual. A Σ-structure is a $\Sigma[\emptyset]$-interpretation. We use x, y, z for first-order variables, p for Boolean ones, and $\bar{x}$, $\bar{y}$, $\bar{z}$, and $\bar{p}$ for tuples of such variables. We also use φ and ψ for formulas, F and G for quantifier-free formulas, $\mathcal{M}$ for interpretations, $\models$ for satisfaction and entailment, $=$ for identity, $\uplus$ for disjoint union, and $\setminus$ for set difference. $FV(\varphi)$ is the set of the variables occurring free in φ. Slightly abusing the notation, $FV(\varphi)$ is also treated as a tuple. Implication is written $\Rightarrow$ and logical equivalence is written $\Leftrightarrow$. If $\mathcal{V}_1 \subseteq \mathcal{V}_2$ (i.e., $\mathcal{V}_1^s \subseteq \mathcal{V}_2^s$ for all $s \in S$), a $\Sigma[\mathcal{V}_2]$-interpretation $\mathcal{M}_2$ is an *extension* of a $\Sigma[\mathcal{V}_1]$-interpretation $\mathcal{M}_1$ to $\mathcal{V}_2$, if $\mathcal{M}_2$ interprets the variables in $\mathcal{V}_2^s \setminus \mathcal{V}_1^s$ for all $s \in S$ and is otherwise identical to $\mathcal{M}_1$.

A theory $\mathcal{T}$ is defined by a signature Σ and a set of Σ-sentences called $\mathcal{T}$-axioms. A model of $\mathcal{T}$, or $\mathcal{T}$-model, is a Σ-structure that satisfies the $\mathcal{T}$-axioms. A $\mathcal{T}[\mathcal{V}]$-model is a $\Sigma[\mathcal{V}]$-interpretation that is a $\mathcal{T}$-model when the interpretation of variables is ignored. A theory $\mathcal{T}$ is *complete*, if it is consistent, and for all Σ-sentences F, either F or $\neg F$ is provable from the $\mathcal{T}$-axioms. In this paper we deal with a single theory $\mathcal{T}$ that has a unique $\mathcal{T}$-model $\mathcal{M}_0$, so that the interpretation of everything except variables is fixed. Therefore $\mathcal{T}$ is complete, for Σ-sentences $\mathcal{T}$-validity, $\mathcal{T}$-satisfiability, and truth in $\mathcal{M}_0$ coincide, all $\mathcal{T}[\mathcal{V}]$-models are extensions of $\mathcal{M}_0$, and a $\mathcal{T}$-satisfiability procedure is concerned only with assignments to variables. Since there are one theory and one signature, we write formula for $\Sigma[\mathcal{V}]$-formula and model for $\mathcal{T}$-model or $\mathcal{T}[\mathcal{V}]$-model. A *conservative theory extension* $\mathcal{T}^+$ of $\mathcal{T}$ adds to Σ special constants, called *values*, to name elements in the domain of $\mathcal{M}_0$ as needed. Conservative means that a $\mathcal{T}$-satisfiable formula is also $\mathcal{T}^+$-satisfiable.

The *quantified SMA problem* for theory $\mathcal{T}$ asks whether $\mathcal{M}_0 \models \varphi$ for an arbitrary formula φ and an initial assignment of values to the variables in $FV(\varphi)$. Formulas have the form $\varphi = \exists\bar{x}.F[\bar{z}, \bar{x}, \bar{p}]\{p_i \leftarrow \exists\bar{y}_i.G_i[\bar{z}, \bar{x}, \bar{y}_i]\}_{i=1}^{k}$ described in the introduction, where $FV(\varphi) = \bar{z}$ and quantified variables are standardized apart. If $FV(\varphi) = \emptyset$, we still have SMA problems when considering subformulas under an assignment to existentially quantified variables.

3 The QSMA Framework

The QSMA algorithm works with a tree representation of a formula φ. A node n in the tree is labeled with a pair $(\bar{x}, F)$, where $\bar{x}$ is a tuple of first-order variables, called the *local variables* of n, and F is a quantifier-free formula. The local variables are implicitly existentially quantified: they are existentially quantified variables whose quantifers have been stripped, so that they are locally free, so to speak, and can be assigned by the algorithm. An arc from a node n to a child node b is labeled with a Boolean variable p. This Boolean variable stands as a *proxy* for the quantified subformula represented by the subtree rooted at node b. Therefore, the Boolean variable p is also considered a proxy of b itself.

A formula φ may have free variables $FV(\varphi) = \bar{z}$, whose assignment is given initially as part of the SMA problem instance. These variables are called *rigid*, because their assignments do not change during the tree traversal. As the algorithm traverses the tree, the local variables of a node n are *rigid* from the point of view of a child node b: their assignments do not change during the traversal of the subtree rooted at b. Therefore, we represent a formula φ as a pair formed by a tuple of rigid variables and a labeled tree. Slightly abusing the terminology, we call this pair a QSMA-*tree*. The root of a tree T is denoted $root(T)$.

Definition 1 **(QSMA-tree).** *Given* $\varphi = \exists\bar{x}.F[\bar{z},\bar{x},\bar{p}]\{p_i \leftarrow \exists\bar{y}_i.G_i[\bar{z},\bar{x},\bar{y}_i]\}_{i=1}^{k}$, *where* $FV(\varphi) = \bar{z}$ *and* $\varphi_i = \exists\bar{y}_i.G_i[\bar{z},\bar{x},\bar{y}_i]$, $1 \leq i \leq k$, *the* QSMA-tree *for* φ *is the pair* $\mathcal{G} = (\bar{z}, T)$, *where* $\bar{z}$ *is called the tuple of the rigid variables of* $\mathcal{G}$, *and* T *is a labeled tree defined inductively as follows:*

- *If* $k = 0$, T *consists of a single node* r *labeled* $(\bar{x}, F[\bar{z},\bar{x}])$;
- *If* $k > 0$, *for all* i, $1 \leq i \leq k$, *let* $\mathcal{G}_i = ((\bar{z},\bar{x}), T_i)$ *be the* QSMA-*tree for* φ_i, *where* $root(T_i)$ *is a node* b_i *labeled* $(\bar{y}_i, G_i[\bar{z},\bar{x},\bar{y}_i])$. *Then* T *is the tree with a new node* r *labeled* $(\bar{x}, F[\bar{z},\bar{x},\bar{p}])$ *as root,* k *outgoing arcs labeled* $p_1, \ldots, p_k$, *and* $b_1, \ldots, b_k$ *as children.*

If subformula φ_i occurs more than once in φ, the same proxy variable p_i is used for all occurrences. The *ancestors* of a node n in T are the nodes on the unique path from $root(T)$ to n excluding n itself. If node n in T is labeled $(\bar{x}, F)$, its k outgoing arcs are labeled $p_1, \ldots, p_k$, and $\bar{x}_1, \ldots, \bar{x}_m$ are the local variables of the ancestors of n, then $FV(F) \subseteq \{\bar{z}, \bar{x}_1, \ldots, \bar{x}_m, \bar{x}, p_1, \ldots, p_k\}$. The set of the *assignable variables at node* n is $Var(n) = \bar{x} \uplus \{p_1, \ldots, p_k\}$. The set of the *rigid variables at node* n is $Rigid(n) = \bar{z} \uplus \bar{x}_1 \uplus \ldots \uplus \bar{x}_m$. Thus, $FV(F) \subseteq Rigid(n) \cup Var(n)$, $Rigid(root(T)) = \bar{z}$, and the QSMA-subtree rooted at node n is $\mathcal{G}_n = (Rigid(n), T_n)$. For a node n with label $(\bar{x}, F)$, the components of the label are denoted $n.\bar{x}$ and $n.F$. The label of the arc from n to a child b is denoted $b.p$.

Example 1. Given $\exists x.((\forall y_1.F_1[x,y_1]) \Rightarrow (\forall y_2.F_2[x,y_2]))$ from Sect. 1.1, let $\varphi = \exists x.((\exists y_1.\neg F_1[x,y_1]) \vee (\neg\exists y_2.\neg F_2[x,y_2])) = \exists x.(p_1 \vee \neg p_2)\{p_i \leftarrow \exists y_i.\neg F_i[x,y_i]\}_{i=1}^{2}$. The QSMA-tree for φ has root r labeled $(x, p_1 \vee \neg p_2)$ with left child b_1 labeled $(y_1, \neg F_1[x,y_1])$, right child b_2 labeled $(y_2, \neg F_2[x,y_2])$, and arcs from r to b_1 and from r to b_2 labeled p_1 and p_2, respectively. Note how $FV(r.F) \subseteq \{x, p_1, p_2\}$, $Var(r) = \{x, p_1, p_2\}$, and $Rigid(r) = \emptyset$. Also, $FV(b_1.F) \subseteq \{x, y_1\}$, $FV(b_2.F) \subseteq \{x, y_2\}$, $Var(b_1) = \{y_1\}$, $Var(b_2) = \{y_2\}$, and $Rigid(b_1) = Rigid(b_2) = \{x\}$.

Example 2. Consider $\forall x.((\exists y_1.(x \simeq 2{\cdot}y_1)) \Rightarrow (\exists y_2.(3{\cdot}x \simeq 2{\cdot}y_2)))$. A double negation eliminates the $\forall$, yielding $\neg(\exists x.((\exists y_1.(x \simeq 2{\cdot}y_1)) \wedge (\forall y_2.(3{\cdot}x \not\simeq 2{\cdot}y_2))))$.

Again, a double negation eliminates the ∀, producing $\neg(\exists x.((\exists y_1.(x \simeq 2{\cdot}y_1)) \wedge (\neg(\exists y_2.(3{\cdot}x \simeq 2{\cdot}y_2)))))$. Let $\varphi = \exists x.((\exists y_1.(x \simeq 2{\cdot}y_1)) \wedge (\neg(\exists y_2.(3{\cdot}x \simeq 2{\cdot}y_2)))) = \exists x.(p_1 \wedge \neg p_2)\{p_1 \leftarrow \exists y_1.(x \simeq 2{\cdot}y_1),\ p_2 \leftarrow \exists y_2.(3{\cdot}x \simeq 2{\cdot}y_2)\}$. The original formula is true in LRA iff φ is false in LRA. The QSMA-tree for φ has root r labeled $(x, p_1 \wedge \neg p_2)$ with left child b_1 labeled $(y_1, x \simeq 2{\cdot}y_1)$, right child b_2 labeled $(y_2, 3{\cdot}x \simeq 2{\cdot}y_2)$, and arcs from r to b_1 and from r to b_2 labeled p_1 and p_2, respectively. The variable sets of this tree are as in Example 1.

Conversely, given a QSMA-tree $\mathcal{G} = (\bar{z}, T)$, we can associate a formula $n.\psi$ to any node n in T and hence to the QSMA-subtree $\mathcal{G}_n = (\mathit{Rigid}(n), T_n)$.

Definition 2 (Formula at a node). *Given a* QSMA*-tree* $\mathcal{G} = (\bar{z}, T)$*, for all nodes* n *of* T*, the* formula $n.\psi$ at node n *is defined inductively as follows:*

- *If* n *is a leaf labeled* $(\bar{x}, F[\bar{z}, \bar{x}])$*, then* $n.\psi = \exists\bar{x}.F[\bar{z}, \bar{x}]$*;*
- *If* n *has label* $(\bar{x}, F[\bar{z}, \bar{x}, \bar{p}])$ *and outgoing arcs labeled* $p_1, \ldots, p_k$*,* $k > 0$*, connecting* n *to children* $b_1, \ldots, b_k$*, let* $b_1.\psi, \ldots, b_k.\psi$ *be the formulas at* $b_1, \ldots, b_k$*. Then* $n.\psi = \exists\bar{x}.F[\bar{z}, \bar{x}, \bar{p}]\{p_i \leftarrow b_i.\psi\}_{i=1}^{k}$*.*

If $\mathcal{G} = (\bar{z}, T)$ is the QSMA-tree for φ and $r = \mathit{root}(T)$, then $r.\psi = \varphi$.

Example 3. For the QSMA-tree in Example 2, $b_1.\psi = \exists y_1.(x \simeq 2{\cdot}y_1)$, $b_2.\psi = \exists y_2.(3{\cdot}x \simeq 2{\cdot}y_2)$, and $r.\psi = \exists x.(p_1 \wedge \neg p_2)\{p_1 \leftarrow \exists y_1.(x \simeq 2{\cdot}y_1),\ p_2 \leftarrow \exists y_2.(3{\cdot}x \simeq 2{\cdot}y_2)\} = \exists x.((\exists y_1.(x \simeq 2{\cdot}y_1)) \wedge \neg(\exists y_2.(3{\cdot}x \simeq 2{\cdot}y_2))) = \varphi$.

Since the input formula φ is represented as a QSMA-tree $\mathcal{G} = (\bar{z}, T)$, the problem of satisfying φ becomes the problem of satisfying $\mathcal{G}$. Therefore, we define *satisfaction of a* QSMA*-tree* next. Slightly abusing the notation, we use $\models$ also for satisfaction of QSMA-trees.

Definition 3 (Satisfaction of a QSMA-tree). *Given a* QSMA*-tree* $\mathcal{G} = (\bar{z}, T)$ *with* $r = \mathit{root}(T)$*, and an extension* $\mathcal{M}$ *of* $\mathcal{M}_0$ *to* $\mathit{Rigid}(r) = \bar{z}$*,* $\mathcal{M} \models \mathcal{G}$ *if there exists an extension* $\mathcal{M}'$ *of* $\mathcal{M}$ *to* $\mathit{Var}(r)$ *such that (i)* $\mathcal{M}' \models r.F$*, and (ii) for all children* b *of* r*,* $\mathcal{M}'(b.p) = \mathsf{true}$ *iff* $\mathcal{M}' \models \mathcal{G}_b$*.*

The QSMA algorithm works by traversing the QSMA-tree $\mathcal{G} = (\bar{z}, T)$, and at each node n in T it assigns the assignable variables in $\mathit{Var}(n) = \bar{x} \uplus \{p_1, \ldots, p_k\}$. This assignment corresponds to the extension $\mathcal{M}'$ in Definition 3. Let b be a child of n: the Boolean variable $b.p$ labeling the arc from n to b is a proxy for the quantified subformula $b.\psi$ of the formula $n.\psi$. If $\mathcal{M}'(b.p) = \mathsf{true}$, the aim of the algorithm is to show that $b.\psi$ is true, and if $\mathcal{M}'(b.p) = \mathsf{false}$, the aim is to show that $b.\psi$ is false. Therefore Condition (ii) in Definition 3 says $\mathcal{M}' \models \mathcal{G}_b$ if $\mathcal{M}'(b.p) = \mathsf{true}$ and $\mathcal{M}' \not\models \mathcal{G}_b$ if $\mathcal{M}'(b.p) = \mathsf{false}$. The next theorem shows that satisfying a formula φ and satisfying the QSMA-tree for φ correspond.

Theorem 1. *For all formulas* φ *with* $FV(\varphi) = \bar{z}$*, for all models* $\mathcal{M}$ *extending* $\mathcal{M}_0$ *to* $\bar{z}$*, if* $\mathcal{G}$ *is the* QSMA*-tree for* φ *then* $\mathcal{M} \models \mathcal{G}$ *iff* $\mathcal{M} \models \varphi$*.*

Checking whether $\mathcal{M} \models \mathcal{G}$ by testing all possible extensions $\mathcal{M}'$ would not do, because for most theories (e.g., LRA) there is an infinite number of extensions.

We need a way to weed out large parts of the space of candidate models. Let $[\![\varphi]\!]$ denote the set of φ's models. We introduce *under-approximations* and *over-approximations* of φ in order to under-approximate and over-approximate $[\![\varphi]\!]$.

Definition 4 (Under- and over-approximation). *Let φ be a formula with $FV(\varphi) = \bar{z}$. Quantifier-free formulas U and O with $FV(U) = FV(O) = \bar{z}$ are, respectively, an* under-approximation *and an* over-approximation *of φ, if for all extensions $\mathcal{M}$ of $\mathcal{M}_0$ to $\bar{z}$, $\mathcal{M} \models U$ implies $\mathcal{M} \models \varphi$ and $\mathcal{M} \models \varphi$ implies $\mathcal{M} \models O$.*

It follows that $[\![U]\!] \subseteq [\![\varphi]\!] \subseteq [\![O]\!]$. Let $\mathcal{G} = (\bar{z}, T)$ be the QSMA-tree for φ, and U and O under- and over-approximations of φ, respectively. Then, $\mathcal{M} \models U$ implies $\mathcal{M} \models \varphi$ which implies $\mathcal{M} \models \mathcal{G}$ by Theorem 1. Thus, satisfying an under-approximation is a sufficient condition to have a solution. On the other hand, $\mathcal{M} \models \neg O$ implies $\mathcal{M} \not\models \varphi$ which implies $\mathcal{M} \not\models \mathcal{G}$ by Theorem 1. By the contrapositive, if $\mathcal{M} \models \mathcal{G}$ then $\mathcal{M} \not\models \neg O$, that is, $\mathcal{M} \models O$. Thus, satisfying an over-approximation is a necessary condition to have a solution. In order to construct such approximations, we assume to have a solver for theory $\mathcal{T}$ (and model $\mathcal{M}_0$) offering:

- *Model extension*: A function SMA such that for all formulas $\exists\bar{x}.F[\bar{z},\bar{x}]$, where $F[\bar{z},\bar{x}]$ is quantifier-free, and all extensions $\mathcal{M}$ of $\mathcal{M}_0$ to $\bar{z}$, $\mathsf{SMA}(F[\bar{z},\bar{x}],\mathcal{M})$ returns either an extension $\mathcal{M}'$ of $\mathcal{M}$ to $\bar{x}$ such that $\mathcal{M}' \models F[\bar{z},\bar{x}]$, or *nil* if there is no such extension.
- *Model-based under-approximation*: A function MBU such that for all formulas $\exists\bar{x}.F[\bar{z},\bar{x}]$, where $F[\bar{z},\bar{x}]$ is quantifier-free, and all extensions $\mathcal{M}$ of $\mathcal{M}_0$ to $\bar{z}$ such that $\mathcal{M} \models \exists\bar{x}.F[\bar{z},\bar{x}]$, $\mathsf{MBU}(F[\bar{z},\bar{x}],\bar{x},\mathcal{M})$ returns a quantifier-free formula $U[\bar{z}]$ such that $\mathcal{M} \models U[\bar{z}]$ and $\mathcal{T} \models U[\bar{z}] \Rightarrow (\exists\bar{x}.F[\bar{z},\bar{x}])$.
- *Model-based over-approximation*: A function MBO such that for all formulas $\exists\bar{x}.F[\bar{z},\bar{x}]$, where $F[\bar{z},\bar{x}]$ is quantifier-free, and all extensions $\mathcal{M}$ of $\mathcal{M}_0$ to $\bar{z}$ such that $\mathcal{M} \not\models \exists\bar{x}.F[\bar{z},\bar{x}]$, $\mathsf{MBO}(F[\bar{z},\bar{x}],\bar{x},\mathcal{M})$ returns a quantifier-free formula $O[\bar{z}]$ such that $\mathcal{M} \not\models O[\bar{z}]$ and $\mathcal{T} \models (\exists\bar{x}.F[\bar{z},\bar{x}]) \Rightarrow O[\bar{z}]$.

MBU and MBO produce, respectively, an under-approximation and an over-approximation. Formula $U[\bar{z}]$ is true in model $\mathcal{M}$ and implies $\exists\bar{x}.F[\bar{z},\bar{x}]$, and hence can be seen as an *interpolant between model and formula*. It was called *model generalization* [12,17], because $U[\bar{z}]$ may have other models in addition to $\mathcal{M}$. Formula $O[\bar{z}]$ follows from $\exists\bar{x}.F[\bar{z},\bar{x}]$ and is false in $\mathcal{M}$, and hence can be seen as a *reverse interpolant between formula and model*, called *model interpolant* [17].

4 The QSMA Algorithm and Its Total Correctness

Let $\mathcal{G} = (\bar{z}, T)$ be the QSMA-tree for input formula φ with $FV(\varphi) = \bar{z}$. Given a model $\mathcal{M}$ extending $\mathcal{M}_0$ to $\bar{z}$, the QSMA algorithm determines whether $\mathcal{M} \models \mathcal{G}$. Suppose that U and O are under- and over-approximations of φ, respectively. Picture $[\![U]\!]$, $[\![\varphi]\!]$, and $[\![O]\!]$ as bubbles. The $[\![U]\!]$ bubble is inside the $[\![\varphi]\!]$ bubble, which is inside the $[\![O]\!]$ bubble. The idea of the algorithm is to zoom in on a model of φ, by progressively weakening U, so that the $[\![U]\!]$ bubble inflates, and progressively strengthening O, so that the $[\![O]\!]$ bubble deflates. The algorithm operates in this manner for all subformulas of φ: for all nodes n of T it maintains under and over-approximations $n.U$ and $n.O$ of $n.\psi$, progressively weakening $n.U$ and strengthening $n.O$. The weakening of $n.U$ is done by introducing a disjunction with an MBU. The strengthening of $n.O$ is done by introducing a conjunction with an MBO. The goal is that $\mathcal{M}$ satisfies $n.U \vee \neg n.O$. As soon as $\mathcal{M}$ satisfies $n.U$, we know that $\mathcal{M} \models \mathcal{G}_n$. As soon as $\mathcal{M}$ satisfies $\neg n.O$, we know that $\mathcal{M} \not\models \mathcal{G}_n$.

```
@pre: $\mathcal{G} = (\bar{z}, T)$: QSMA-tree for $\varphi$ with $FV(\varphi) = \bar{z}$; $\mathcal{M}$: extension of $\mathcal{M}_0$ to $\bar{z}$
@post: $rv$ iff $\mathcal{M} \models \mathcal{G}$ ($rv$ is "returned value")
function QSMA($\mathcal{M}$, $T$)
    for all nodes $n$ in $T$ do
        $n.U \leftarrow \bot$
        $n.O \leftarrow \top$
    return subtreeIsSolved($root(T)$, $\mathcal{M}$)
```

Fig. 1. Pseudocode of the main function of the QSMA algorithm

The main function QSMA (Fig. 1) initializes $n.U$ to $\bot$ (under-approximation of all formulas and identity for disjunction) and $n.O$ to $\top$ (over-approximation of all formulas and identity for conjunction) for all nodes n of T. Then QSMA calls the function `subtreeIsSolved` (Fig. 2) with arguments $root(T)$ and $\mathcal{M}$.

Function `subtreeIsSolved` takes a node n and a model $\mathcal{M}$ extending $\mathcal{M}_0$ to $Rigid(n)$ and determines whether $\mathcal{M} \models \mathcal{G}_n$. If $\mathcal{M} \models n.U$ it returns *true*; if $\mathcal{M} \models \neg n.O$ it returns *false* (lines 3–5 in Fig. 2). Otherwise (i.e., $\mathcal{M} \models \neg n.U \wedge n.O$), it enters a loop whose body contains the following steps:

1. Build a formula L as the conjunction of $n.F$ and a formula for every child b of n, denoted $n \rightarrow b$ (line 7 in Fig. 2). The shape of the formula for b is better explained by considering a model of L and hence in the next step.
2. Invoke the SMA function to search for an extension $\mathcal{M}'$ of $\mathcal{M}$ to $Var(n)$ such that $\mathcal{M}' \models L$ (line 8). For all children b of n, $b.p \in Var(n)$ and $\mathcal{M}'$ assigns a Boolean value to $b.p$. If $\mathcal{M}'(b.p) = \mathsf{true}$, the subformula for b in L reduces to $b.O$, so that $\mathcal{M}' \models L$ implies $\mathcal{M}' \models b.O$. Since QSMA seeks to satisfy $b.\psi$ and $[\![b.\psi]\!] \subseteq [\![b.O]\!]$, it starts at least from a model of $b.O$. If $\mathcal{M}'(b.p) = \mathsf{false}$, the subformula for b in L reduces to $\neg b.U$, so that $\mathcal{M}' \models L$ implies $\mathcal{M}' \models \neg b.U$. Since QSMA seeks to falsify $b.\psi$ and $[\![b.U]\!] \subseteq [\![b.\psi]\!]$, it starts at least from a model of $\neg b.U$. The proof of partial correctness[1] of `subtreeIsSolved` shows that the existence of an $\mathcal{M}'$ such that $\mathcal{M}' \models L$ is necessary for $\mathcal{M} \models \mathcal{G}_n$.

[1] See https://mariapaola.github.io/CDSATandQSMA.html for a copy of this paper with the proofs inserted.

```
@pre: $\mathcal{M}$: extension of $\mathcal{M}_0$ to $\mathit{Rigid}(n)$, and $I = \forall b \in T.\ [\![b.U]\!] \subseteq [\![b.\psi]\!] \subseteq [\![b.O]\!]$
@post: $I$ and $\mathcal{M} \models (n.U \vee \neg n.O)$ and ($rv$ iff $\mathcal{M} \models \mathcal{G}_n$) and ($rv$ iff $\mathcal{M} \models n.U$) and ($\neg rv$ iff $\mathcal{M} \models \neg n.O$)
 1: function subtreeIsSolved($n$, $\mathcal{M}$)
 2:     if $\mathcal{M} \models n.U$ then
 3:         return $\mathit{true}$
 4:     else if $\mathcal{M} \models \neg n.O$ then
 5:         return $\mathit{false}$
 6:     while true do
 7:         $L \leftarrow n.F \wedge \bigwedge_{n \to b}((b.p \Rightarrow b.O) \wedge (\neg b.p \Rightarrow \neg b.U))$
 8:         $\mathcal{M}' \leftarrow \mathsf{SMA}(L, \mathcal{M})$
 9:         if $\mathcal{M}' = \mathit{nil}$ then
10:             $n.O \leftarrow n.O \wedge \mathsf{MBO}(L, FV(L) \setminus \mathit{Rigid}(n), \mathcal{M})$
11:             return $\mathit{false}$
12:         else
13:             if solutionForallChildren($n$, $\mathcal{M}'$) then
14:                 $L' \leftarrow nF \wedge \bigwedge_{n \to b}((b.p \Rightarrow b.U) \wedge (\neg b.p \Rightarrow \neg b.O))$
15:                 $n.U \leftarrow n.U \vee \mathsf{MBU}(L', FV(L') \setminus \mathit{Rigid}(n), \mathcal{M})$
16:                 return $\mathit{true}$
17:
18: function solutionForallChildren($n$, $\mathcal{M}$)
19:     for all children $b$ of $n$ do
20:         if $\mathcal{M}(b.p) \neq$ subtreeIsSolved($b$, $\mathcal{M}$) then
21:             return $\mathit{false}$
22:     return $\mathit{true}$
```

Fig. 2. Pseudocode of the auxiliary functions of the QSMA algorithm

3. If SMA returns *nil*, then $\mathcal{M} \not\models \mathcal{G}_n$; `subtreeIsSolved` updates $n.O$ to its conjunction with $\mathsf{MBO}(L, FV(L) \setminus \mathit{Rigid}(n), \mathcal{M})$ (line 10). Since $\mathcal{M} \not\models L$, by MBO's specification we know that $\mathcal{M} \not\models \mathsf{MBO}(L, FV(L) \setminus \mathit{Rigid}(n), \mathcal{M})$. This update ensures that $\mathcal{M} \not\models n.O$, so that $\mathcal{M} \models \neg n.O$. Then `subtreeIsSolved` returns *false* (line 11).
4. Otherwise, we have an extension $\mathcal{M}'$ that satisfies L and hence $n.F$, so that there is potential for $\mathcal{M} \models \mathcal{G}_n$. Function `solutionForallChildren` is invoked to determine whether this is the case.
5. The function `solutionForallChildren` calls `subtreeIsSolved` for every child b of n. As soon as it finds a child b such that $\mathcal{M}(b.p) = \mathsf{true}$ and the call `subtreeIsSolved`(b,$\mathcal{M}$) returns *false*, or $\mathcal{M}(b.p) = \mathsf{false}$ and the call `subtreeIsSolved`(b,$\mathcal{M}$) returns *true*, it returns *false*, because it found a QSMA-subtree where candidate model $\mathcal{M}$ fails. If this does not happen, `solutionForallChildren` returns *true*.
6. If `solutionForallChildren` returns *true*, `subtreeIsSolved` builds a formula L' as the conjunction of $n.F$ and a formula for every child b of n (line 14). If $\mathcal{M}'(b.p) = \mathsf{true}$, the subformula for b in L' reduces to $b.U$. If $\mathcal{M}'(b.p) = \mathsf{false}$, the subformula for b in L' reduces to $\neg b.O$. The proof of partial correctness of `subtreeIsSolved` shows that $\mathcal{M}' \models L'$ and that $\mathcal{M}' \models L'$ is a sufficient condition for $\mathcal{M} \models \mathcal{G}_n$. Then `subtreeIsSolved` updates $n.U$ to its disjunction with $\mathsf{MBU}(L', FV(L') \setminus \mathit{Rigid}(n), \mathcal{M})$ (line 15). Since $\mathcal{M}' \models L'$,

by MBU's specification we know that $\mathcal{M}' \models \mathsf{MBU}(L', FV(L') \setminus Rigid(n), \mathcal{M})$. This update ensures that $\mathcal{M}' \models n.U$. Then `subtreeIsSolved` returns *true* (line 16).

7. If `solutionForallChildren` returns *false*, the control returns to line 7. Suppose that `solutionForallChildren` returned *false*, because it found a child b of n such that $\mathcal{M}(b.p) = \mathsf{true}$ and `subtreeIsSolved`$(b,\mathcal{M})$ returned *false*. Then the call `subtreeIsSolved`$(b,\mathcal{M})$ updated the formula $b.O$ (line 10). Suppose that `solutionForallChildren` returned *false*, because it found a child b of n such that $\mathcal{M}(b.p) = \mathsf{false}$ and `subtreeIsSolved`$(b,\mathcal{M})$ returned *true*. Then the call `subtreeIsSolved`$(b,\mathcal{M})$ updated the formula $b.U$ (line 15). Either way the state has changed, variable L gets a new formula on line 7, and the subsequent call to SMA will not produce the same model.

Example 4. Apply `subtreeIsSolved` to the root of the QSMA-tree in Example 1. Formula L gets $p_1 \vee \neg p_2$. SMA produces an $\mathcal{M}'$ that assigns values to x, p_1, and p_2. Suppose that $\mathcal{M}'$ satisfies $p_1 \vee \neg p_2$ by assigning true to p_1. In the recursive call on b_1, formula L gets $\neg F_1[x, y_1]$. If SMA produces an $\mathcal{M}''$ that extends $\mathcal{M}'$ with an assignment to y_1 such that $\mathcal{M}'' \models \neg F_1[x, y_1]$, we have a model. Suppose that $\mathcal{M}'$ satisfies $p_1 \vee \neg p_2$ by assigning false to p_2. In the recursive call on b_2, formula L gets $\neg F_2[x, y_2]$. If SMA fails to produces an $\mathcal{M}''$ that extends $\mathcal{M}'$ with an assignment to y_2 such that $\mathcal{M}'' \models \neg F_2[x, y_2]$, we have a model.

Theorem 2. *The function* `subtreeIsSolved` *is partially correct: if the preconditions hold and the function halts, then the postconditions hold.*

For termination, we begin with the MBU and MBO functions. Let $\mathcal{T}$ be LRA with a theory extension LRA^+ that adds constant symbols $\tilde{q}$ for all rational numbers q. Consider an MBU function such that $\mathsf{MBU}(F[\bar{z}, x], x, \mathcal{M}) = F[\bar{z}, x]\{x \leftarrow \tilde{q}\}$ and $\mathcal{M} \models F[\bar{z}, \tilde{q}]$. This kind of MBU is called *generalization-by-substitution* [12]. While $F[\bar{z}, \tilde{q}]$ is an under-approximation of $\exists x. F[\bar{z}, x]$, this MBU is not a good choice for termination. By applying MBU repeatedly with an infinite enumeration of rational constants, the QSMA algorithm could build an infinite sequence of under-approximations $(\bigvee_{i=1}^{n} F[\bar{z}, x]\{x \leftarrow \tilde{q}_i\})_{n \in \mathbb{N}}$ none of which is LRA-equivalent to $\exists x. F[\bar{z}, x]$. The next definition excludes such MBU functions, by requiring that for a given formula and variable tuple (that depends on the formula), MBU can generate only finitely many formulas.

Definition 5 (Finite basis). *An* MBU *function has* finite basis *if the set* $\{\mathsf{MBU}(F[\bar{z}, \bar{x}], \bar{x}, \mathcal{M}) \mid \mathcal{M} : \textit{extension of } \mathcal{M}_0 \textit{ to } \bar{z} \textit{ such that } \mathcal{M} \models \exists \bar{x}. F[\bar{z}, \bar{x}]\}$ *is finite for all quantifier-free formulas* $F[\bar{z}, \bar{x}]$ *and tuples* $\bar{x}$.

The notion of an MBO function having a finite basis is defined in the same way with $\not\models$ in place of $\models$.

Lemma 1. *If* MBU *and* MBO *have finite basis, for all (possibly infinite) series of calls* $\{\texttt{subtreeIsSolved}(n, \mathcal{M}_i)\}_i$*, all satisfying the preconditions and all terminating, formulas* $n.U$ *and* $n.O$ *are updated only a finite number of times.*

Once nontermination due to MBU or MBO is excluded even for an infinite series of halting calls, termination is proved by induction on the QSMA-tree.

Theorem 3. *If the* MBU *and* MBO *functions have finite basis, whenever the preconditions are satisfied the function* `subtreeIsSolved` *halts.*

Example 5. Apply `subtreeIsSolved` to the root of the QSMA-tree in Example 2. Formula L gets $p_1 \wedge \neg p_2$. SMA produces an $\mathcal{M}'$ that assigns values to x, p_1, and p_2. Suppose that $\mathcal{M}'$ assigns 1 to x, while it must assign true to p_1 and false to p_2. In the recursive call on b_1, formula L gets $x \simeq 2{\cdot}y_1$. If SMA produces an $\mathcal{M}''$ that extends $\mathcal{M}'$ with $y_1 {\leftarrow} \frac{1}{2}$, we have a model of $\mathcal{G}_{b_1}$. In the recursive call on b_2, formula L gets $3{\cdot}x \simeq 2{\cdot}y_2$. If SMA produces an $\mathcal{M}''$ that extends $\mathcal{M}'$ with $y_2 {\leftarrow} \frac{3}{2}$, we have a model of $\mathcal{G}_{b_2}$, but because $\mathcal{M}'(p_2) = \mathsf{false}$, there is no model of $\mathcal{G}$. Indeed, formula φ of Example 2 is false as the original formula is true.

5 The **OptiQSMA** Algorithm and Its Total Correctness

YicesQS implements an optimized variant of QSMA, called $\mathsf{OptiQSMA}$, that reduces the number of recursive calls to `subtreeIsSolved` by entrusting more work to each call to SMA. Reconsider the behavior of QSMA in Example 4. We can avoid a recursive call to `subtreeIsSolved` by asking SMA to satisfy $(p_1 \vee \neg p_2) \wedge (p_1 \Rightarrow \neg F_1[x, y_1])$ in lieu of $p_1 \vee \neg p_2$. This way, if the candidate model returned by SMA assigns true to p_1, it also assigns to x and y_1 values that satisfy $\neg F_1[x, y_1]$. This means that $\exists y_1 . \neg F_1[x, y_1]$ is found true without recursion. On the other hand, if p_2 is assigned false, the algorithm still has to make the recursive call to see if it can satisfy $\exists y_2 . \neg F_2[x, y_2]$.

The idea of $\mathsf{OptiQSMA}$ is to do a look-ahead on a path in the QSMA-tree, doing the work in one shot rather then through recursive calls on all the nodes in the path. The look-ahead applies to a path such that the Boolean labels of all the arcs in the path are assigned true by the candidate model. The following definition builds a formula to allow the look-ahead.

Definition 6 (Look-ahead formula). *Given a* QSMA*-tree* $\mathcal{G} = (\bar{z}, T)$*, for all nodes* n *of* T *the* look-ahead formula *of* n *is* $LF(n) = n.F \wedge \bigwedge_{n \rightarrow b} (b.p \Rightarrow LF(b))$.

The next definition distinguishes the nodes that are handled together in one shot without recursion and those where recursion is still needed. Nodes of the first kind are called *no alternation nodes*, because such nodes are on a path as described above, where all Boolean labels are assigned true and hence there is no alternation between true and false. Nodes of the second kind are called *first alternation nodes*, because they are the nodes reached by the first arc whose Boolean label is assigned false.

```
@pre: $\mathcal{G} = (\bar{z}, T)$: QSMA-tree for $\varphi$ with $FV(\varphi) = \bar{z}$; $\mathcal{M}$: extension of $\mathcal{M}_0$ to $\bar{z}$
@post: $rv$ iff $\mathcal{M} \models_{la} \mathcal{G}$
function OptiQSMA($\mathcal{M}$, $T$)
    for all nodes $n$ in $T$ do
        $n.U \leftarrow \bot$
    ans $\leftarrow$ optiSubtreeIsSolved($root(T)$, $\mathcal{M}$)
    if ans = SAT(_) then
        return $true$
    else if ans = UNSAT(_) then
        return $false$
```

Fig. 3. Pseudocode of the main function of the OptiQSMA algorithm

Definition 7 (No alternation nodes and first alternation nodes). *Given a* QSMA*-tree* $\mathcal{G} = (\bar{z}, T)$ *for all nodes* n *of* T *and extensions* $\mathcal{M}$ *of* $\mathcal{M}_0$ *to* $FV(LF(n))$*, the set* $\mathsf{NAN}(n, \mathcal{M})$ *of the* no-alternation nodes from n according to $\mathcal{M}$ *(resp. the set* $\mathsf{FAN}(n, \mathcal{M})$ *of the* first-alternation nodes from n according to $\mathcal{M}$*) contains all and only the nodes* b *such that: (i)* b *is a descendant of* n *through a path* $n \rightarrow n_1 \rightarrow \ldots \rightarrow n_q \rightarrow b$ *(*$q \geq 0$*), (ii)* $\forall i,\ 1 \leq i \leq q$*,* $\mathcal{M}(n_i.p) = \mathsf{true}$*, and (iii)* $\mathcal{M}(b.p) = \mathsf{true}$ *(resp.* $\mathcal{M}(b.p) = \mathsf{false}$*).*

A node $b \in \mathsf{FAN}(n, \mathcal{M})$ such that $q = 0$ in Condition (i) of Definition 7 is a child of n: for a child there is no optimization. The OptiQSMA algorithm seeks a candidate model $\mathcal{M}$ that satisfies $LF(n)$ and recurses only on the nodes in $\mathsf{FAN}(n, \mathcal{M})$. Therefore, the definition of *satisfaction with look-ahead*, denoted $\models_{la}$, follows the pattern of Definition 3, replacing $r.F$ with $LF(r)$ and Condition (ii) of Definition 3 with a condition for the nodes in the FAN set.

Definition 8 (Satisfaction with look-ahead). *Given a* QSMA*-tree* $\mathcal{G} = (\bar{z}, T)$ *with* $r = root(T)$ *and an extension* $\mathcal{M}$ *of* $\mathcal{M}_0$ *to* $Rigid(r) = \bar{z}$*,* $\mathcal{M} \models_{la} \mathcal{G}$ *if there exists an extension* $\mathcal{M}'$ *of* $\mathcal{M}$ *to* $FV(LF(r))$ *such that (i)* $\mathcal{M}' \models LF(r)$ *and (ii) for all nodes* $b \in \mathsf{FAN}(r, \mathcal{M}')$*,* $\mathcal{M}' \not\models_{la} \mathcal{G}_b$*.*

Since for the nodes $b \in \mathsf{FAN}(r, \mathcal{M}')$ it is $\mathcal{M}'(b.p) = \mathsf{false}$, the $\models_{la}$ relation is negated in Condition (ii). The next theorem shows that the optimization does not change the problem.

Theorem 4. *Given a* QSMA*-tree* $\mathcal{G} = (\bar{z}, T)$ *and an extension* $\mathcal{M}$ *of* $\mathcal{M}_0$ *to* $\bar{z}$*,* $\mathcal{M} \models \mathcal{G}$ *if and only if* $\mathcal{M} \models_{la} \mathcal{G}$*.*

The OptiQSMA algorithm maintains under-approximations $n.U$ of $n.\psi$ for all nodes n, but not over-approximations. Accordingly, the main function OptiQSMA (Fig. 3) initializes only $n.U$ for all nodes n, and then calls `optiSubtreeIsSolved` (Fig. 4). This function returns $\mathsf{SAT}(U)$ if $\mathcal{M} \models_{la} \mathcal{G}$ and $\mathsf{UNSAT}(O)$ if $\mathcal{M} \not\models_{la} \mathcal{G}$. The formula U is an under-approximation of $r.\psi$ ($r = root(T)$) such that $\mathcal{M} \models U$. The formula O is an over-approximation of $r.\psi$ such that $\mathcal{M} \not\models O$. The main function OptiQSMA has no usage for U and O and merely returns *true* or *false* accordingly. Function `optisubtreeIsSolved` builds and returns under-approximations and over-approximations recursively. The reason for saving only

```
@pre: $\mathcal{M}$ is an extension of $\mathcal{M}_0$ to $Rigid(n)$, and $I = \forall b \in T.\ [\![b.U]\!] \subseteq [\![b.\psi]\!]$
@post: $I$ and
$\{rv = \mathsf{UNSAT}(O)$ implies $[(\forall b \in T.\ [\![b.\psi]\!] \subseteq [\![O]\!])$ and $\mathcal{M} \not\models O]\}$ and
$\{rv = \mathsf{SAT}(U)$ implies $[(\forall b \in T.\ [\![b.U]\!] \subseteq [\![b.\psi]\!])$ and $\mathcal{M} \models U]\}$
 1: function optiSubtreeIsSolved($n$, $\mathcal{M}$)
 2:     while true do
 3:         $L \leftarrow LF(n) \wedge \bigwedge_{n \to^+ b}(\neg b.p \Rightarrow \neg b.U)$
 4:         $\mathcal{M}' \leftarrow \mathsf{SMA}(L, \mathcal{M})$
 5:         if $\mathcal{M}' = nil$ then
 6:             return $\mathsf{UNSAT}(\mathsf{MBO}(L, FV(L) \setminus Rigid(n), \mathcal{M}))$
 7:         else
 8:             reasons $\leftarrow \top$
 9:             if solutionForallDescendants($n$, $\mathcal{M}'$, reasons) then
10:                 $L' \leftarrow LF(n) \wedge$ reasons
11:                 return $\mathsf{SAT}(\mathsf{MBU}(L', FV(L') \setminus Rigid(n), \mathcal{M}))$
12:
13: function solutionForallDescendants($n$, $\mathcal{M}$, reasons)
14:     for all $b \in \mathsf{FAN}(n, \mathcal{M})$ do
15:         ans $\leftarrow$ optiSubtreeIsSolved($b$, $\mathcal{M}$)
16:         if ans $= \mathsf{SAT}(U)$ then
17:             $b.U \leftarrow b.U \vee U$
18:             return $false$
19:         else if ans $= \mathsf{UNSAT}(O)$ then
20:             reasons $\leftarrow$ reasons $\wedge (\neg b.p \Rightarrow \neg O)$
21:     for all $b \in \mathsf{NAN}(n, \mathcal{M})$ do
22:         reasons $\leftarrow$ reasons $\wedge\ b.p$
23:     return $true$
```

Fig. 4. Pseudocode of the auxiliary functions of the optiQSMA algorithm

under-approximations is practical, and will become clear after the illustration of `optisubtreeIsSolved`. This function takes a node n and a model $\mathcal{M}$ extending $\mathcal{M}_0$ to $Rigid(n)$ and determines whether $\mathcal{M} \models_{la} \mathcal{G}_n$, by executing a loop whose body contains the following steps:

1. Build a formula L (line 3 in Fig. 4) as the conjunction of the look-ahead formula $LF(n)$ (in lieu of $n.F$ in line 7 of Fig. 2) and a formula for every descendant b of n, denoted $n \to^+ b$ (in lieu of child as in Fig. 2).
2. Invoke the SMA function to search for an extension $\mathcal{M}'$ of $\mathcal{M}$ to $Var(n)$ such that $\mathcal{M}' \models L$. For those descendants b for which $\mathcal{M}'(b.p) = \mathsf{false}$, the subformula for b in L reduces to $\neg b.U$ as in Step 2 of the description of `subtreeIsSolved`. For those descendants b for which $\mathcal{M}'(b.p) = \mathtt{true}$, the subformula for b in L reduces to `true`, in agreement with the fact that over-approximations are not kept.
3. If SMA returns nil, `optiSubtreeIsSolved` returns $\mathsf{UNSAT}(O)$, where O is simply the outcome of applying MBO to L and $\mathcal{M}$, as over-approximations

are not kept. Otherwise, there is potential for satisfaction with look-ahead. Function `optiSubtreeIsSolved` initializes the formula `reasons` to $\top$ and invokes `solutionForallDescendants` passing `reasons` by reference.

4. Function `solutionForallDescendants` considers first all descendants b in $\mathsf{FAN}(n, \mathcal{M})$, and calls `optiSubtreeIsSolved`$(b, \mathcal{M})$ for each of them. If this call returns $\mathsf{SAT}(U)$, it means that $\mathcal{M} \models_{la} \mathcal{G}_b$; `solutionForallDescendants` weakens $b.U$ by disjunction with U and returns *false*.
   If `optiSubtreeIsSolved`$(b, \mathcal{M})$ returns $\mathsf{UNSAT}(O)$, it means that $\mathcal{M} \not\models_{la} \mathcal{G}_b$, and we move on to the next descendant in $\mathsf{FAN}(n, \mathcal{M})$. Prior to that, `reasons` is strengthened by conjunction with $\neg b.p \Rightarrow \neg O$. For all descendants b in $\mathsf{NAN}(n, \mathcal{M})$, `solutionForallDescendants` strengthens `reasons` by conjunction with $b.p$.
5. If `solutionForallDescendants` returns *true*, `optiSubtreeIsSolved` builds formula L' as $LF(n) \wedge$`reasons`, and returns $\mathsf{SAT}(U)$, where U is the outcome of the application of MBU to L' and $\mathcal{M}$. Otherwise, the control returns to line 3. Since `solutionForallDescendants` returned *false*, it means that it found a node b in $\mathsf{FAN}(n, \mathcal{M})$ for which `optiSubtreeIsSolved`$(b,\mathcal{M})$ returned $\mathsf{SAT}(U)$ and the formula $b.U$ was updated (line 17). Therefore the state has changed, variable L gets a new formula on line 3, and the subsequent call to SMA will not produce the same model.

In the experiments it turned out that storing over-approximations for all nodes is less efficient than using them to compute L' and then forget them. Thus, the over-approximation O encapsulated in the $\mathsf{UNSAT}(O)$ value returned by a recursive call to `optiSubtreeIsSolved` is used to build the temporary formula `reasons`, but it is not saved, and `reasons` is used to compute L'.

Theorem 5. *The function* `optiSubtreeIsSolved` *is partially correct: if the preconditions hold and the function halts, then the postconditions hold.*

The proof of partial correctness of `optiSubtreeIsSolved` shows that every model that satisfies $L' = (LF(n) \wedge$ `reasons`$)$ fulfills Definition 8. In this sense, `reasons` is an explanation of why a model is found with look-ahead.

Theorem 6. *If the* MBU *and* MBO *functions have finite basis, whenever the preconditions are satisfied the function* `optiSubtreeIsSolved` *halts.*

6 The YicesQS Solver and Experimental Results

The $\mathsf{OptiQSMA}$ algorithm is implemented in YicesQS to equip Yices 2 with support for quantifiers for complete theories (unrelated to Yices 2 support for quantifiers in UF).[2] MBO is available as model interpolation from Yices's MCSAT [10] solver for quantifier-free formulas, including theory-specific techniques for bitvectors (BV) [15] and arithmetic. The latter are based on NLSAT [16] and ultimately on Cylindrical Algebraic Decomposition (CAD). Basic MBU is done

[2] See https://github.com/disteph/yicesQS and https://yices.csl.sri.com/.

BV.

CVC5	854/970	25,584s
Q3B	835/970	13,510s
Z3	775/970	7,712s
Bitwuzla	759/970	15,572s
Q3B-pBDD	754/970	15,553s
YicesQS	708/970	3,862s
Ultim.Elim.	304/970	4,204s

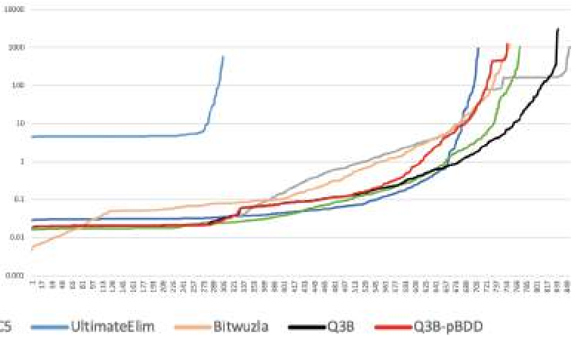


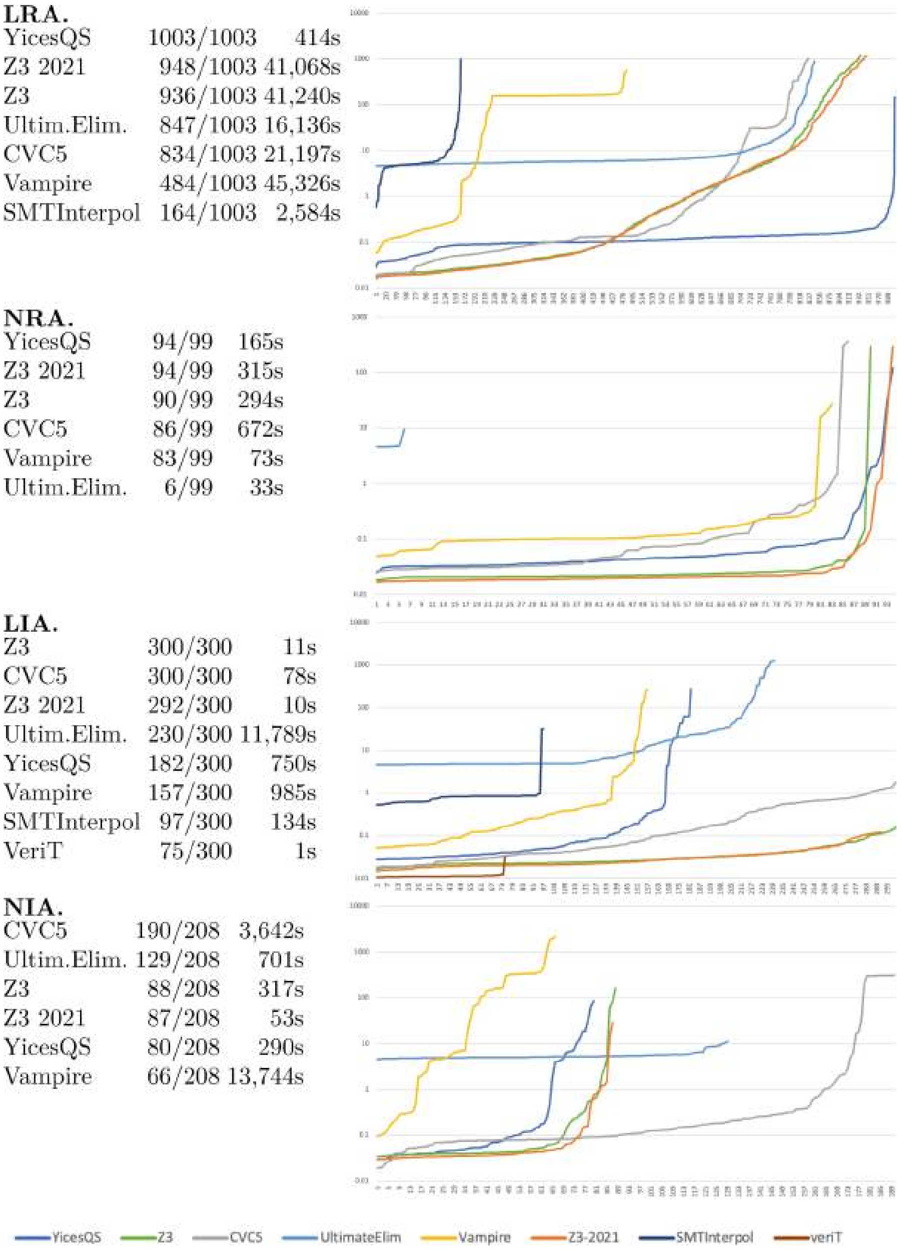
Fig. 5. Plot for BV.

as generalization-by-substitution [12] and improved with *model-based projection* (e.g., [18]) for arithmetic, and *invertibility conditions* [21], including ϵ-terms, for BV. In YicesQS model-based projection also is based on CAD.

In the 2022 SMT competition, YicesQS entered the single-query, non-incremental tracks of BV, LRA, LIA, NRA, and NIA (nonlinear integer arithmetic). The experiments were run on the Starexec cluster with a 20 min timeout per benchmark and 60GB of memory. The benchmarks were a subset of the SMT-LIB collection. The results presented below were computed by running the competition script `join.sh` on the raw data from StarExec,[3] sorting the data, and producing the plots that are available online.[4] A description of the participating solvers can be found on the competition website.[5]

Figure 5 shows the results for BV, where YicesQS solved quickly a high number of benchmarks (compared for example with CVC5), but was not outstanding, possibly because YicesQS 2022 makes a limited use of invertibility conditions for model interpolation. Figure 6 shows the results for the four arithmetics. The columns on the left list number of solved instances and time to solve them for each logic and solver. In the plot on the right, each color corresponds to a solver and point (x, y) of that color means that the x^{th} fastest-solved benchmark was solved by that solver in time y (log scale). 2021 Z3 is included because in some of these logics it performed slightly better than 2022 Z3. The logic where YicesQS performed best is LRA: it was the only solver to solve all 1,003 benchmarks. Z3 2021 was second best, solving 948 benchmarks with a total runtime about 100 times higher. YicesQS has neither a special treatment (e.g., simplex-based) of linear problems, nor integer-specific techniques: it relies on CAD-based techniques for MBU and MBO also for integer problems. Thus, it is somewhat average on LIA and NIA. These two theories are undecidable (NRA due to division by 0) and hence they lie outside of the theoretical framework of QSMA. YicesQS

[3] https://github.com/SMT-COMP/smt-comp/tree/master/2022/results.
[4] http://www.csl.sri.com/users/sgl/Work/Cade2023-data/index.html.
[5] https://smt-comp.github.io/2022/participants.html.



LRA.

YicesQS	1003/1003	414s
Z3 2021	948/1003	41,068s
Z3	936/1003	41,240s
Ultim.Elim.	847/1003	16,136s
CVC5	834/1003	21,197s
Vampire	484/1003	45,326s
SMTInterpol	164/1003	2,584s

NRA.

YicesQS	94/99	165s
Z3 2021	94/99	315s
Z3	90/99	294s
CVC5	86/99	672s
Vampire	83/99	73s
Ultim.Elim.	6/99	33s

LIA.

Z3	300/300	11s
CVC5	300/300	78s
Z3 2021	292/300	10s
Ultim.Elim.	230/300	11,789s
YicesQS	182/300	750s
Vampire	157/300	985s
SMTInterpol	97/300	134s
VeriT	75/300	1s

NIA.

CVC5	190/208	3,642s
Ultim.Elim.	129/208	701s
Z3	88/208	317s
Z3 2021	87/208	53s
YicesQS	80/208	290s
Vampire	66/208	13,744s

Fig. 6. Plots for the four arithmetics.

answers should still be correct, but termination can be lost. With Z3 being a non-competing participant in the SMT 2022 competition, YicesQS came second for *Largest Contribution* (single queries), because of its overall performance in the four arithmetics, where it also came first for satisfiable instances and in the 24 sec timeout setup (instead of 20 min).

7 Discussion: Related Work and Future Work

Quantified SMT was approached by a procedure with an ∃-solver and a ∀-solver for prenex normal form formulas with ∃∀ prefix [12]. A formulation as a game between an ∃-player and a ∀-player appeared with the *QSAT algorithm* [3] for prenex normal form formulas with $(\exists\forall)^+$ prefix. QSMA accepts arbitrary formulas with quantifiers in arbitrary positions.

Both QSAT and QSMA work for a generic theory $\mathcal{T}$ over basic $\mathcal{T}$-specific components. QSAT uses *model-based projection* [3,18] and a solver for quantifier-free satisfiability that supports UNSAT cores. Model-based projection is an instance of MBU. An UNSAT core (as a conjunction) is an MBO in the special case where the input assignment is Boolean. While MBO can produce UNSAT cores, MBO generalizes the concept of UNSAT core with theory-specific reasoning when there are *non-Boolean input assignments*, as it is the case in QSMA. It is unclear whether the combination of UNSAT cores and theory-specific MBU can emulate MBO or provide the same benefits. QSAT is implemented in Z3 and it is the default solver for LIA, LRA, and NRA.

YicesQS is a recent implementation that only participated in the SMT competition in 2021 and 2022. Directions for further development include augmenting integer reasoning, and improving model interpolation in BV by a better usage of invertibility conditions. Another lead for future work is to compose QSMA within the *CDSAT framework for conflict-driven reasoning in unions of theories* [4–6]. For this, one may need to drop the assumption that there is a unique model $\mathcal{M}_0$ and only its extensions need to be considered, which will be a generalization also in the single theory case. As most known MBU and MBO functions are for single theories, one may have to study how to get MBU and MBO functions for a union of theories from such functions for the component theories. Another issue is the interplay between QSMA's recursive descent over the QSMA-tree for the formula and CDSAT's conflict-driven search.

Acknowledgements. Part of this work was done while the first and third authors were visiting SRI International, whose support is much appreciated. This material is based upon work supported by NSF with awards CCF-1816936 and CCF-1817204. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the US Government or NSF.

References

1. Althaus, E., Kruglov, E., Weidenbach, C.: Superposition modulo linear arithmetic SUP(LA). In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS (LNAI), vol. 5749, pp. 84–99. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04222-5_5

2. Baumgartner, P., Waldmann, U.: Hierarchic superposition with weak abstraction. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 39–57. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_3
3. Bjørner, N., Janota, M.: Playing with quantified satisfaction (short paper). In: Fehnker, A., McIver, A., Sutcliffe, G., Voronkov, A. (eds.) Short Presentations at LPAR-20. EPiC Series in Computing, vol. 35, pp. 15–27. EasyChair (2015)
4. Bonacina, M.P., Graham-Lengrand, S., Shankar, N.: Conflict-driven satisfiability for theory combination: transition system and completeness. J. Autom. Reason. **64**(3), 579–609 (2020). https://doi.org/10.1007/s10817-018-09510-y
5. Bonacina, M.P., Graham-Lengrand, S., Shankar, N.: CDSAT for nondisjoint theories with shared predicates: arrays with abstract length. In: Hyvärinen, A., Déharbe, D. (eds.) Proceedings of the SMT-20. CEUR Proceedings, vol. 3185, pp. 18–37. CEUR WS-org (2022)
6. Bonacina, M.P., Graham-Lengrand, S., Shankar, N.: Conflict-driven satisfiability for theory combination: lemmas, modules, and proofs. J. Autom. Reason. **66**(1), 43–91 (2022). https://doi.org/10.1007/s10817-021-09606-y
7. Bonacina, M.P., Lynch, C.A., de Moura, L.: On deciding satisfiability by theorem proving with speculative inferences. J. Autom. Reason. **47**(2), 161–189 (2011). https://doi.org/10.1007/s10817-010-9213-y
8. Bradley, A.R., Manna, Z.: The Calculus of Computation - Decision Procedures with Applications to Verification. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74113-8
9. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_13
10. de Moura, L., Jovanović, D.: A model-constructing satisfiability calculus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 1–12. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_1
11. Detlefs, D.L., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005). https://doi.org/10.1145/1066100.1066102
12. Dutertre, B.: Solving exists/forall problems with Yices. In: Proceedings of the SMT-13 (2015)
13. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 167–182. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_12
14. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25
15. Graham-Lengrand, S., Jovanović, D., Dutertre, B.: Solving bitvectors with MCSAT: explanations from bits and pieces. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12166, pp. 103–121. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_7
16. Jovanović, D., de Moura, L.: Solving non-linear arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 339–354. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_27
17. Jovanović, D., Dutertre, B.: Interpolation and model checking for nonlinear arithmetic. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 266–288. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_13

18. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods Syst. Design **48**(3), 175–205 (2016). https://doi.org/10.1007/s10703-016-0249-4
19. Korovin, K., Voronkov, A.: Integrating linear arithmetic into superposition calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74915-8_19
20. Moskal, M.: Fx7 or in software, it is all about quantifiers. System Descriptions at SMT-COMP (2007). http://smtcomp.cs.uiowa.edu/2007/descriptions/fx7.pdf
21. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Solving quantified bit-vectors using invertibility conditions. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 236–255. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_16
22. Reynolds, A., Tinelli, C., de Moura, L.: Finding conflicting instances of quantified formulas in SMT. In: Claessen, K., Kuncak, V. (eds.) Proceedings of the FMCAD 2014, pp. 195–202. ACM and IEEE (2014). https://dl.acm.org/doi/10.5555/2682923.2682957

Uniform Substitution for Dynamic Logic with Communicating Hybrid Programs

Marvin Brieger[1(✉)], Stefan Mitsch[2], and André Platzer[2,3]

[1] LMU Munich, Munich, Germany
marvin.brieger@sosy.ifi.lmu.de
[2] Carnegie Mellon University, Pittsburgh, USA
smitsch@cs.cmu.edu, platzer@kit.edu
[3] Karlsruhe Institute of Technology, Karlsruhe, Germany

Abstract. This paper introduces a uniform substitution calculus for dL_{CHP}, the dynamic logic of communicating hybrid programs. Uniform substitution enables parsimonious prover kernels by using axioms instead of axiom schemata. Instantiations can be recovered from a single proof rule responsible for soundness-critical instantiation checks rather than being spread across axiom schemata in side conditions. Even though communication and parallelism reasoning are notorious for necessitating subtle soundness-critical side conditions, uniform substitution when generalized to dL_{CHP} manages to limit and isolate their conceptual overhead. Since uniform substitution has proven to simplify the implementation of hybrid systems provers substantially, uniform substitution for dL_{CHP} paves the way for a parsimonious implementation of theorem provers for hybrid systems with communication and parallelism.

Keywords: Uniform substitution · Parallel programs · Differential dynamic logic · Assumption-commitment reasoning · CSP

1 Introduction

$$\frac{[\alpha]\varphi \qquad [\beta]\psi}{[\alpha \parallel \beta](\varphi \wedge \psi)} \;(\star\star)$$

Fig. 1. The proof rule is only sound under subtle side conditions $(\star\star)$.

Hybrid systems and parallel systems are notoriously subtle to analyze. Combining both not only culminates these subtleties but is further complicated because parallel hybrid systems are interlocked by synchronization in a shared global time. The *dynamic logic of communicating hybrid programs* dL_{CHP} [6] tames the complexity of parallel hybrid systems providing a compositional proof calculus that disentangles reasoning into purely discrete, continuous, and communication pieces. However, the calculus is subject to schematic side conditions whose implementation is generally error-prone causing large soundness-critical code bases [30]. In particular, compositional reasoning about parallelism as in the idealized proof rule in Fig. 1 holds the challenge to exhaustively characterize *all* side conditions required to make *all* instances of this proof rule sound. Proof

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 96–115, 2023.
https://doi.org/10.1007/978-3-031-38499-8_6

systems for discrete parallelism [1,19,27,35,44,46] already have complicated side conditions, but complexity only increases with continuous interactions in shared global time.

In order to compositionally support compositional reasoning for parallel hybrid systems, this paper generalizes Church's uniform substitution [8] and develops a uniform substitution calculus [30–32] for dL_{CHP}. Uniform substitution modularizes the calculus itself enabling its parsimonious implementation. Although applicable to discrete parallelism, the dL_{CHP} development resolves the inherent challenge that parallel hybrid systems always synchronize in time.

Uniform substitution adopts a finite list of concrete formulas as axioms instead of an infinite set of formulas via axiom schemata with side conditions. This enables theorem provers without the extensive algorithmic checks otherwise required for each schema to sort out unsound instances. Thanks to the proof rule US for uniform substitution, only sound instances derive from the axioms such that the parallel composition rule in dL_{CHP} could be adopted almost literally as above, but with all the soundness-critical checking encapsulated solely in rule US. Thanks to US's checking, parallel systems reasoning even reduces to a single parallel injection axiom $[\alpha]\psi \rightarrow [\alpha \parallel \beta]\psi$ that merely describes the preservation of property ψ of one parallel component α in the parallel system $\alpha \parallel \beta$. Proofs about $\alpha \parallel \beta$ reduce to a sequence of property embeddings with this axiom from local abstractions of the subcomponents, which combine soundly due to US.

Soundness checks in uniform substitution are ultimately determined by the binding structures as identified in the static semantics. The development of uniform substitution for dL_{CHP} is, therefore, grounded in the following key observation: Communication and parallelism both cause additional binding structure that needs attention in the substitution process performed by rule US:

(B I) Expressions depend on communication along (co)finite channel sets (besides finitely many free variables), which, by the core substitution principle [8], must not be introduced free into contexts where they are written.

(B II) Subprograms in a parallel context need to be restricted in the variables and channels written as compositional proof rules for parallelism require local abstractions of subprograms not depending on the internals of the context [35].

Grounded in the need for abstraction (B II), $[\alpha]\psi \rightarrow [\alpha \parallel \beta]\psi$ can only be adopted as a sound axiom schema if α and β do not share state, and if program β does not interfere with the contract ψ, i.e., (i) ψ has no free variables bound by β (with exceptions), and (ii) ψ does not depend on communication channels written by β (except for channels joint with α). This extensive side condition would need nontrivial soundness-critical implementations of dL_{CHP} axiom schemata. Still, uniform substitution can be lifted with only small changes locally checking for clashes with written channels, and prohibited variables or channels.

The modularity of uniform substitution is the key to the parsimonious implementation [23] of the theorem prover KeYmaera X [11] for differential dynamic logic dL and differential game logic dGL [29], thus paving the way for a straightforward theorem prover implementation of dL_{CHP}. Since dL_{CHP} conservatively generalizes dL [6], its uniform substitution calculus inherits the complete [33] axiomatic treatment of differential equation invariants [30]. All proofs are in [7].

2 Dynamic Logic of Communicating Hybrid Programs

This section briefly recaps dL_{CHP} [6], the dynamic logic of communicating hybrid programs (CHPs). It combines hybrid programs [28] with CSP-style communication and parallelism [15]. By assumption-commitment (ac) reasoning [22,46,47], dL_{CHP} allows compositional verification of parallelism in dL. For uniform substitution, function and predicate symbols, and program constants are added.

2.1 Syntax

The set of variables $V = V_{\mathbb{R}} \cup V_{\mathbb{N}} \cup V_{\mathcal{T}}$ has real ($V_{\mathbb{R}}$), integer ($V_{\mathbb{N}}$), and trace ($V_{\mathcal{T}}$) variables. For each $x \in V_{\mathbb{R}}$, the differential symbol x' is in $V_{\mathbb{R}}$, too. The designated variable $\mu \in V_{\mathbb{R}}$ represents the shared global time. The set of channel names is Ω. By convention $x, y \in V_{\mathbb{R}}$, $n \in V_{\mathbb{N}}$, $h \in V_{\mathcal{T}}$, $\mathrm{ch} \in \Omega$, and $z \in V$. Channel set $Y \subseteq \Omega$ is (co)finite. Vectorial expressions are denoted $\bar{e}$. Moreover, $f^{\mathbb{M}}$, $g^{\mathbb{M}}$ are $\mathbb{M}$-valued function symbols and p, q, r are predicate symbols, where argument sorts are annotated by $\_ : \mathbb{M}_1, \ldots, \mathbb{M}_k$. Finally, a, b are program constants.

Definition 1 (Terms). *Terms consist of real* ($\mathrm{Trm}_{\mathbb{R}}$)*, integer* ($\mathrm{Trm}_{\mathbb{N}}$)*, channel* ($\mathrm{Trm}_{\Omega}$)*, and trace* ($\mathrm{Trm}_{\mathcal{T}}$) *terms, and are defined by the grammar below, where* $\theta, \theta_1, \theta_2 \in \mathbb{Q}[V_{\mathbb{R}}] \subset \mathrm{Trm}_{\mathbb{R}}$ *are polynomials in* $V_{\mathbb{R}}$*:*

$$\begin{aligned}
\mathrm{Trm}_{\mathbb{R}}: &\quad \eta_1, \eta_2 ::= x \mid f^{\mathbb{R}}(Y, \bar{e}) \mid \eta_1 + \eta_2 \mid \eta_1 \cdot \eta_2 \mid (\theta)' \mid \mathtt{val}(te) \mid \mathtt{time}(te) \\
\mathrm{Trm}_{\mathbb{N}}: &\quad ie_1, ie_2 ::= n \mid f^{\mathbb{N}}(Y, \bar{e}) \mid ie_1 + ie_2 \mid |te| \\
\mathrm{Trm}_{\Omega}: &\quad ce_1, ce_2 ::= f^{\Omega}(Y, \bar{e}) \mid \mathtt{chan}(te) \\
\mathrm{Trm}_{\mathcal{T}}: &\quad te_1, te_2 ::= h \mid f^{\mathcal{T}}(Y, \bar{e}) \mid \langle ch, \theta_1, \theta_2 \rangle \mid te_1 \cdot te_2 \mid te \downarrow Y \mid te[ie]
\end{aligned}$$

Real terms are polynomials in $V_{\mathbb{R}}$ enriched with function symbols $f^{\mathbb{R}}(Y, \bar{e})$ (including constants $c \in \mathbb{Q}$) only depending on communication along channels Y and terms $\bar{e}$, differential terms $(\theta)'$, and $\mathtt{val}(te)$ and $\mathtt{time}(te)$, which access the value and the timestamp of the last communication in te, respectively. By convention, $\theta \in \mathbb{Q}[V_{\mathbb{R}}]$ denotes a pure polynomial in $V_{\mathbb{R}}$ without $(\cdot)'$, $\mathtt{val}(\cdot)$, and $\mathtt{time}(\cdot)$ as they occur in programs. For simplicity, we do not define $\mathbb{Q}[V_{\mathbb{R}}] \subset \mathrm{Trm}_{\mathbb{R}}$ as a fifth term sort but use the convention that function symbols $g^{\mathbb{R}}$ can only be replaced with $\mathbb{Q}[V_{\mathbb{R}}]$-terms. Integer terms are variables n, function symbols $f^{\mathbb{N}}(Y, \bar{e})$ (including constants 0, 1), addition, and length $|te|$ of trace term te.[1] The function symbol $f^{\Omega}(Y, \bar{e})$ includes constants $\mathrm{ch} \in \Omega$, and $\mathtt{chan}(te)$ is channel access. Trace terms record the communication history of programs. They encompass variables h, function symbols $f^{\mathcal{T}}(Y, \bar{e})$ (including the empty trace ϵ), communication items $\langle \mathrm{ch}, \theta_1, \theta_2 \rangle$ with value θ_1 and timestamp θ_2, projection

[1] Omitting multiplication results in decidable Presburger arithmetic [34].

$te \downarrow Y$ onto channels Y, and access $te[ie]$ of the ie-th item in te. Where useful, $\mathsf{op}(\bar{e})$ denotes built-in function symbols of fixed interpretation, e.g., $\cdot + \cdot$.

$\mathsf{dL}_{\mathrm{CHP}}$'s context-sensitive program and formula syntax presumes notions of free and bound variables (Sect. 2.3) defined on the context-free syntax:

Definition 2 (Programs). Communicating hybrid programs *are defined by the following grammar, where* $\theta \in \mathbb{Q}[V_{\mathbb{R}}]$ *is a polynomial in* $V_{\mathbb{R}}$ *and* $\chi \in \mathrm{FOL}_{\mathbb{R}}$ *is a formula of first-order real-arithmetic. In* $\alpha \parallel \beta$*, the subprograms must not share state but can share time and history, i.e.,* $\mathsf{BV}(\alpha) \cap \mathsf{BV}(\beta) \subseteq \{\mu, \mu'\} \cup V_{\mathcal{T}}$*.*[2]

$$\begin{aligned}\alpha, \beta ::= {}& a(\!|Y, \bar{z}|\!) \mid x := \theta \mid x := * \mid ?\chi \mid \{x' = \theta \,\&\, \chi\} \mid \alpha; \beta \mid \alpha \cup \beta \mid \alpha^* \mid \\ & ch(h)!\theta \mid ch(h)?x \mid \alpha \parallel \beta\end{aligned}$$

The program constant $a(\!|Y, \bar{z}|\!)$ restricts the written channels to $Y \subseteq \Omega$ and the bound variables to $\bar{z} \subseteq V_{\mathbb{R}} \cup V_{\mathcal{T}}$, where Y and $\bar{z}$ are (co)finite. Instead of $a(\!|Y, \bar{z}|\!)$, write a if Y and $\bar{z}$ can be arbitrary. Assignment $x := \theta$ updates x to θ, nondeterministic assignment $x := *$ assigns an arbitrary real value to x, and the test $?\chi$ does nothing if χ holds and aborts the computation otherwise. The continuous evolution $\{x' = \theta \& \chi\}$ follows the ODE $x' = \theta$ for any duration as long as formula χ is not violated. The global time μ evolves with every continuous evolution according to ODE $\mu' = 1$. Sequential composition $\alpha; \beta$ executes β after α, choice $\alpha \cup \beta$ executes α or β nondeterministically, α^* repeats α zero or more times, $\mathrm{ch}(h)!\theta$ sends θ along channel ch, and $\mathrm{ch}(h)?x$ receives a value into variable x along channel ch. The trace variable h records communication. Finally, $\alpha \parallel \beta$ executes α and β in parallel synchronized in global time μ.

Example 3. The program $\mathsf{ct}^* \parallel \mathsf{ve}^*$ models a simplified cruise control [24]. The vehicle ve repeatedly receives a target velocity $v_{\mathsf{ve}}^{\mathrm{tr}}$ from the controller ct along channel tar. The target $v_{\mathsf{ct}}^{\mathrm{tr}}$ sent by ct is in range $[0, V]$. Hence, ve's velocity v_{ve} stays in range $[0, V]$ within the $\epsilon > 0$ time units till the next communication if $v_{\mathsf{ve}} \in [0, V]$ held initially. The evolution $\{t' = 1\}$ allows passage of time in ct.

$$\mathsf{ct} \equiv v_{\mathsf{ct}}^{\mathrm{tr}} := *; ?(0 \leq v_{\mathsf{ct}}^{\mathrm{tr}} \leq V); \mathrm{tar}(h)!v_{\mathsf{ct}}^{\mathrm{tr}}; \{t' = 1\}$$

$$\mathsf{ve} \equiv \mathrm{tar}(h)?v_{\mathsf{ve}}^{\mathrm{tr}}; a_{\mathsf{ve}} := \frac{v_{\mathsf{ve}}^{\mathrm{tr}} - v_{\mathsf{ve}}}{\epsilon}; t_0 := \mu; \{v_{\mathsf{ve}}' = a_{\mathsf{ve}} \,\&\, \mu - t_0 \leq \epsilon\}$$

Definition 4 (Formulas). *Formulas are defined by the grammar below for relations* $\sim$*, terms* $e_1, e_2 \in \mathrm{Trm}$ *of equal sort, and* $z \in V$*. Moreover, the ac-formulas are unaffected by state change in* α*, i.e.,* $(\mathsf{FV}(\mathsf{A}) \cup \mathsf{FV}(\mathsf{C})) \cap \mathsf{BV}(\alpha) \subseteq V_{\mathcal{T}}$*.*

$$\varphi, \psi, \mathsf{A}, \mathsf{C} ::= e_1 \sim e_2 \mid p(Y, \bar{e}) \mid \neg\varphi \mid \varphi \wedge \psi \mid \forall z\, \varphi \mid [\alpha]\psi \mid [\alpha]_{\{\mathsf{A},\mathsf{C}\}}\psi$$

The formulas combine first-order dynamic logic with ac-reasoning. Predicate symbols $p(Y, \bar{e})$ depend on channels Y and terms $\bar{e}$. The ac-box $[\alpha]_{\{\mathsf{A},\mathsf{C}\}}\psi$

[2] Previous work [6] disallows reading of variables bound in parallel as their change is not observable. This restriction is conceptually desirable but not soundness-critical. Here we drop it for simplicity, but it could be maintained by US as well.

expresses that C holds after each communication event and ψ in the final state, for all runs of α whose incoming communication satisfies A. Other connectives $\vee$, $\rightarrow$, $\leftrightarrow$ and quantifiers $\exists z\,\varphi \equiv \neg\forall z\,\neg\varphi$ can be derived. The relations $\sim$ include $=$ for all term sorts, $\geq$ on real and integer terms, and prefixing $\preceq$ on trace terms.

By convention, the predicate symbol $q_{\mathbb{R}}$ can only be replaced with formulas of first-order real arithmetic. It serves as placeholder for tests χ in CHPs.

Example 5. The cruise control from Example 3 is safe if its velocity stays in range $[0, V]$. This can be expressed with the formula $\varphi \rightarrow [\mathtt{ct}^* \parallel \mathtt{ve}^*]\psi_{\text{safe}}$, where $\psi_{\text{safe}} \equiv 0 \leq v_{\mathtt{ve}} \leq V$ and $\varphi \equiv \psi_{\text{safe}} \wedge \epsilon > 0 \wedge V > 0$.

2.2 Semantics

A *trace* $\tau = (\tau_1, ..., \tau_k)$ is a finite chronological sequence of communication events $\tau_i = \langle \text{ch}_i, d_i, s_i \rangle$, where $\text{ch}_i \in \Omega$, and $d_i \in \mathbb{R}$ is the communicated value, and $s_i \in \mathbb{R}$ is a timestamp such that $s_i \leq s_j$ for $1 \leq i < j \leq k$. A *recorded trace* $\tau = (\tau_1, ..., \tau_k)$ additionally carries a trace variable $h_i \in V_{\mathcal{T}}$ with each event, i.e., $\tau_i = \langle h_i, \text{ch}_i, d_i, s_i \rangle$. For variable $z \in V_{\mathbb{M}}$ and $\mathbb{M} \in \{\mathbb{R}, \mathbb{N}, \mathcal{T}\}$, let $\mathit{type}(z) = \mathbb{M}$. A *state* v maps each $z \in V$ to a value $v(z) \in \mathit{type}(z)$. The sets of traces, recorded traces, and states are denoted $\mathcal{T}$, $\mathcal{T}_{\text{rec}}$, and $\mathcal{S}$, respectively.

For $d \in \mathit{type}(z)$, the state v_z^d is the modification of v at z to d. For $\tau \in \mathcal{T}_{\text{rec}}$, the trace $\tau(h) \in \mathcal{T}$ is obtained from the subsequence of τ carrying $h \in V_{\mathcal{T}}$ by removing the carried variable. *State-trace concatenation* $v \cdot \tau \in \mathcal{S}$ for $\tau \in \mathcal{T}_{\text{rec}}$, appends $\tau(h)$ to v at h for all $h \in V_{\mathcal{T}}$. The *projection* $\tau \downarrow Y$ of (recorded) trace τ is the subsequence of all communication events in τ whose channel is in $Y \subseteq \Omega$. The *state projection* $v \downarrow Y \in \mathcal{S}$ modifies v at h to $v(h) \downarrow Y$ for all $h \in V_{\mathcal{T}}$.

An *interpretation* I assigns a function $I(f^{\mathbb{M}} : \mathbb{M}_1, \ldots, \mathbb{M}_k) : \bigtimes_{i=1}^{k} \mathbb{M}_i \rightarrow \mathbb{M}$ to each function symbol $f^{\mathbb{M}}$ that is smooth in all real-valued arguments if $\mathbb{M} = \mathbb{R}$, and a relation $I(p : \mathbb{M}_1, \ldots, \mathbb{M}_k) \subseteq \bigtimes_{i=1}^{k} \mathbb{M}_i$ to each k-ary predicate symbol p.

Definition 6 (Term Semantics). *The* valuation $Iv[\![e]\!] \in \mathbb{R} \cup \mathbb{N} \cup \Omega \cup \mathcal{T}$ *of term e in interpretation I and state v is defined as follows:*

$$\begin{aligned}
Iv[\![z]\!] &= v(z) \\
Iv[\![f(Y, e_1, ..., e_k)]\!] &= I(f)(I\tilde{v}[\![e_1]\!], ..., I\tilde{v}[\![e_k]\!]) && \text{where}\, \tilde{v} = v \downarrow Y \\
Iv[\![\mathsf{op}(e_1, \ldots, e_k)]\!] &= \mathsf{op}(Iv[\![e_1]\!], \ldots, Iv[\![e_k]\!]) && \text{for builtin}\, \mathsf{op} \in \{\cdot + \cdot, \cdot \downarrow Y, \ldots\} \\
Iv[\![(\theta)']\!] &= \sum_{x \in V_{\mathbb{R}}} v(x') \frac{\partial Iv[\![\theta]\!]}{\partial x}
\end{aligned}$$

The projection $\tilde{v} = v \downarrow Y$ ensures that $f(Y, \bar{e})$ only depends on Y, i.e., the communication in v along channels $Y^{\complement}$ does not matter. The differentials $(\theta)'$ have a semantics describing the local rate of change of θ [30].

The denotational semantics of CHPs [6] combines dL's Kripke semantics [30] with a linear history semantics [47] and a global notion of time. Denotations are subsets of $\mathcal{D} = \mathcal{S} \times \mathcal{T}_{\text{rec}} \times \mathcal{S}_{\perp}$ with $\mathcal{S}_{\perp} = \mathcal{S} \cup \{\perp\}$. Final state $\perp$ marks an unfinished computation, i.e., it still can be continued or was aborted due to a failing test. If ($w' = \bot$ and $\tau' \preceq \tau$), where $\preceq$ is the prefix relation on traces, or $(\tau', w') = (\tau, w)$, then (τ', w') is a prefix of (τ, w) written $(\tau', w') \preceq (\tau, w)$. Since (even empty) communication of unfinished computations is still observable, denotations $D \subseteq \mathcal{D}$ of CHPs are prefix-closed and total, i.e., $(v, \tau, w) \in D$ and $(\tau', w') \preceq (\tau, w)$ implies $(v, \tau', w') \in D$, and $\bot_{\mathcal{D}} \subseteq D$ with $\bot_{\mathcal{D}} = \mathcal{S} \times \{\epsilon\} \times \{\bot\}$. Moreover, all $(v, \tau, w) \in D$ are chronological, i.e., $v(\mu) \leq w(\mu)$ and when $\tau = (\tau_1, \ldots, \tau_k) \neq \epsilon$ and let $\tau_i(\mu) = (\langle h_i, \mathrm{ch}_i, d_i, s_i \rangle)(\mu) = s_i$, then $v(\mu) \leq \tau_1(\mu)$ and if $w \neq \bot$, then $\tau_k(\mu) \leq w(\mu)$. Note that τ is chronological as all traces are.

The interpretation $I(a(\!|Y, \bar{z}|\!)) \subseteq \mathcal{D}$ of a program constant $a(\!|Y, \bar{z}|\!)$ is a prefix-closed and total set of chronological computations that (i) only communicate along (write) channels Y and (ii) only bind variables $\bar{z}$. More precisely, for all $(v, \tau, w) \in I(a(\!|Y, \bar{z}|\!))$, we have (i) $\tau \downarrow Y^{\complement} = \epsilon$, and (ii) $v = w$ on $V_{\mathcal{T}}$ and $w \cdot \tau = v$ on $\bar{z}^{\complement}$. For $D, M \subseteq \mathcal{D}$, we define $D_{\bot} = \{(v, \tau, \bot) \mid (v, \tau, w) \in D\}$, and $(v, \tau, w) \in D \triangleright M$ if $(v, \tau_1, u) \in D$ and $(u, \tau_2, w) \in M$ exist with $\tau = \tau_1 \cdot \tau_2$. For states w_α, w_β, the merged state $w_\alpha \oplus w_\beta$ is $\bot$ if one of the substates w_α or w_β is $\bot$. Otherwise, $w_\alpha \oplus w_\beta = w_\alpha$ on $\mathsf{BV}(\alpha)$ and $w_\alpha \oplus w_\beta = w_\beta$ on $\mathsf{BV}(\alpha)^{\complement}$ (or, equivalently by syntactic well-formedness, on $\mathsf{BV}(\beta)^{\complement}$ and $\mathsf{BV}(\beta)$, respectively). If Y is the set of all channel names occurring in α, we write $\tau \downarrow \alpha$ for $\tau \downarrow Y$.

Definition 7 (Program semantics). *Given an interpretation I, the* semantics *$I[\![\alpha]\!] \subseteq \mathcal{D}$ of a CHP α is defined as follows, where $\bot_{\mathcal{D}} = \mathcal{S} \times \{\epsilon\} \times \{\bot\}$ and $\vDash$ denotes the satisfaction relation (Definition 8):*

$$I[\![a(\!|Y, \bar{z}|\!)]\!] = I(a(\!|Y, \bar{z}|\!))$$

$$I[\![x := \theta]\!] = \bot_{\mathcal{D}} \cup \{(v, \epsilon, w) \mid w = v_x^d \textit{ where } d = Iv[\![\theta]\!]\}$$

$$I[\![x := *]\!] = \bot_{\mathcal{D}} \cup \{(v, \epsilon, w) \mid w = v_x^d \textit{ where } d \in \mathbb{R}\}$$

$$I[\![?\chi]\!] = \bot_{\mathcal{D}} \cup \{(v, \epsilon, v) \mid Iv \vDash \chi\}$$

$$\begin{aligned} I[\![\{x' = \theta \,\&\, \chi\}]\!] = \bot_{\mathcal{D}} \cup \big\{&(v, \epsilon, \varphi(s)) \mid v = \varphi(0) \textit{ on } \{\mu', x'\}^{\complement}, \textit{ and } \varphi(\zeta) = \varphi(0) \\ &\textit{on } \{x, x', \mu, \mu'\}^{\complement}, \textit{ and } I\varphi(\zeta) \vDash \mu' = 1 \wedge x' = \theta \wedge \chi \textit{ for all } \zeta \in [0, s] \textit{ and} \\ &\textit{a solution } \varphi : [0, s] \to \mathcal{S} \textit{ with } \varphi(\zeta)(z') = \frac{d\varphi(t)(z)}{dt}(\zeta) \textit{ for } z \in \{x, \mu\}\big\} \end{aligned}$$

$$I[\![ch(h)!\theta]\!] = \{(v, \tau, w) \mid (\tau, w) \preceq (\langle h, ch, d, v(\mu)\rangle, v) \textit{ where } d = Iv[\![\theta]\!]\}$$

$$I[\![ch(h)?x]\!] = \{(v, \tau, w) \mid (\tau, w) \preceq (\langle h, ch, d, v(\mu)\rangle, v_x^d) \textit{ where } d \in \mathbb{R}\}$$

$$I[\![\alpha \cup \beta]\!] = I[\![\alpha]\!] \cup I[\![\beta]\!]$$

$$I[\![\alpha; \beta]\!] = I[\![\alpha]\!] \,\hat{\circ}\, I[\![\beta]\!] \overset{\text{def}}{=} (I[\![\alpha]\!])_{\bot} \cup (I[\![\alpha]\!] \triangleright I[\![\beta]\!])$$

$$I[\![\alpha^*]\!] = \bigcup_{n \in \mathbb{N}} (I[\![\alpha]\!])^n = \bigcup_{n \in \mathbb{N}} I[\![\alpha^n]\!] \quad \text{where } \alpha^0 \equiv ?\mathsf{T} \text{ and } \alpha^{n+1} = \alpha; \alpha^n$$

$$I[\![\alpha_1 \parallel \alpha_2]\!] = \left\{ (v, \tau, w_{\alpha_1} \oplus w_{\alpha_2}) \;\middle|\; \begin{array}{l} (v, \tau \downarrow \alpha_j, w_{\alpha_j}) \in I[\![\alpha_j]\!] \textit{ for } j = 1, 2, \textit{ and} \\ w_{\alpha_1} = w_{\alpha_2} \textit{ on } \{\mu, \mu'\}, \textit{ and } \tau = \tau \downarrow (\alpha_1 \| \alpha_2) \end{array} \right\}$$

The semantics is indeed constructed prefix-closed, total, and chronological. Communication τ of $\alpha_1 \parallel \alpha_2$ is implicitly characterized via its subsequences for the subprograms. By $\tau = \tau \downarrow (\alpha_1 \parallel \alpha_2)$, there is no non-causal communication. Joint communication and the whole computation are synchronized in global time by the projections and by $w_{\alpha_1} = w_{\alpha_2}$ on $\{\mu, \mu'\}$, respectively. Likewise, by projection, communication is synchronously recorded by trace variables.

Definition 8 (Formula semantics). *The* satisfaction $Iv \vDash \phi$ *of a* dL_{CHP} *formula* ϕ *in interpretation* I *and state* v *is inductively defined as follows:*

1. $Iv \vDash e_1 {\sim} e_2$ *if* $Iv[\![e_1]\!] \sim Iv[\![e_2]\!]$ where $\sim$ is any relation symbol
2. $Iv \vDash p(Y, e_1, \ldots, e_k)$ *if* $(I\tilde{v}[\![e_1]\!], \ldots, I\tilde{v}[\![e_k]\!]) \in I(p)$ where $\tilde{v} = v \downarrow Y$
3. $Iv \vDash \varphi \wedge \psi$ *if* $Iv \vDash \varphi$ *and* $Iv \vDash \psi$
4. $Iv \vDash \neg\varphi$ *if* $Iv \nvDash \varphi$*, i.e., it is not the case that* $Iv \vDash \varphi$
5. $Iv \vDash \forall z\, \varphi$ *if* $Iv_z^d \vDash \varphi$ *for all* $d \in type(z)$
6. $Iv \vDash [\alpha]\psi$ *if* $Iw \cdot \tau \vDash \psi$ *for all* $(v, \tau, w) \in I[\![\alpha]\!]$ *with* $w \neq \bot$
7. $Iv \vDash [\alpha]_{\{\mathsf{A},\mathsf{C}\}}\psi$ *if for all* $(v, \tau, w) \in I[\![\alpha]\!]$ *the following conditions hold:*

$$\{Iv \cdot \tau' \mid \tau' \prec \tau\} \vDash \mathsf{A} \textit{ implies } Iv \cdot \tau \vDash \mathsf{C} \tag{commit}$$

$$\big(\{Iv \cdot \tau' \mid \tau' \preceq \tau\} \vDash \mathsf{A} \textit{ and } w \neq \bot\big) \textit{ implies } Iw \cdot \tau \vDash \psi \tag{post}$$

Where $U \vDash \varphi$ *for a set of interpretation-state pairs* U *and any formula* φ *if* $Iv \vDash \varphi$ *for all* $Iv \in U$*. In particular,* $\emptyset \vDash \varphi$*.*

In item 6 and 7, reachable worlds are built from states v and w, and communication τ, as change of state *and* communication are observable. The strict prefix $\prec$ for the assumption in case (commit) in item 6 excludes (when $\mathsf{A} \equiv \mathsf{C}$) the circularity that commitment C can be shown in states where it is assumed.

2.3 Static Semantics

In the uniform substitution process, checks of free and bound variables, as well as accessed and written channels, separate sound from unsound axiom instantiations. As parallelism requires fine-grained control over channels, the static semantics for dL [30] is lifted to a communication-aware static semantics for dL_{CHP}. It uses accessed channels to characterize the subsequence of a communication trace influencing truth of a formula even more precisely than free variables.

To precisely grasp free and bound variables, and accessed and written channels, Definition 9 gives a semantic characterization. In this section, formulas are considered truth-valued, i.e., $Iv[\![\phi]\!] = \mathbf{tt}$ if $Iv \vDash \phi$ and $Iv[\![\phi]\!] = \mathbf{ff}$ if $Iv \nvDash \phi$.

Definition 9 (Static semantics). *For term or formula* e*, and program* α*, free variables* $\mathsf{FV}(e)$ *and* $\mathsf{FV}(\alpha)$*, bound variables* $\mathsf{BV}(\alpha)$*, accessed channels* $\mathsf{CN}(e)$*, and written channels* $\mathsf{CN}(\alpha)$ *form the static semantics.*

$$\mathsf{FV}(e) = \{z \in V \mid \exists I, v, \tilde{v} \textit{ such that } v = \tilde{v} \textit{ on } \{z\}^{\complement} \textit{ and } Iv[\![e]\!] \neq I\tilde{v}[\![e]\!]\}$$

$$\mathsf{CN}(e) = \{ch \in \Omega \mid \exists I, v, \tilde{v} \textit{ such that } v \downarrow \{ch\}^{\complement} = \tilde{v} \downarrow \{ch\}^{\complement} \textit{ and } Iv[\![e]\!] \neq I\tilde{v}[\![e]\!]\}$$

$$\begin{aligned}\mathsf{FV}(\alpha) = \{z \in V \mid \exists I, v, \tilde{v}, \tau, w \textit{ such that } v = \tilde{v} \textit{ on } \{z\}^{\complement} \textit{ and } (v, \tau, w) \in I[\![\alpha]\!],\\ \textit{and there is no } (\tilde{v}, \tilde{\tau}, \tilde{w}) \in I[\![\alpha]\!] \textit{ such that } \tilde{\tau} = \tau \textit{ and } w = \tilde{w} \textit{ on } \{z\}^{\complement}\}\end{aligned}$$

$$\mathsf{BV}(\alpha) = \{z \in V \mid \exists I, (v, \tau, w) \in I[\![\alpha]\!] \textit{ such that } w \neq \bot \textit{ and } (w \cdot \tau)(z) \neq v(z)\}$$

$$\mathsf{CN}(\alpha) = \{ch \in \Omega \mid \exists I, (v, \tau, w) \in I[\![\alpha]\!] \textit{ such that } \tau \downarrow \{ch\} \neq \epsilon\}$$

The already subtle static semantics of hybrid systems [30] becomes even more subtle with communication and parallelism. For example, CHPs (silently) synchronize with the global time μ, which is free and bound in ODEs, and the differential μ' is bound, i.e.,$\mu \in \mathsf{FV}(\{x' = \theta \,\&\, \chi\})$ and $\mu, \mu' \in \mathsf{BV}(\{x' = \theta \,\&\, \chi\})$ if the evolution has a run of non-zero duration, regardless of whether μ occurs in x. Since reachable worlds of CHPs consist of communication *and* state, bound variables $\mathsf{BV}(\alpha)$ of program α compare v with the state-trace concatenation $w \cdot \tau$ instead of missing τ. Consequently, $h \in \mathsf{BV}(\mathrm{ch}(h)!\theta) \subseteq \mathsf{FV}(\mathrm{ch}(h)!\theta)$, which also reflects that the initial communication never gets lost.

Lemma 10 (Bound effect property). *The sets* $\mathsf{BV}(\alpha)$ *and* $\mathsf{CN}(\alpha)$ *are the smallest sets with the* bound effect property for program α*. That is,* $v = w$ *on* $V_{\mathcal{T}}$ *and* $v = w \cdot \tau$ *on* $\mathsf{BV}(\alpha)^{\complement}$ *if* $w \neq \bot$*, and* $\tau \downarrow \mathsf{CN}(\alpha)^{\complement} = \epsilon$ *for all* $(v, \tau, w) \in I[\![\alpha]\!]$*.*

By the following *communication-aware* coincidence property, terms and formulas only depend on their free variables, which for trace variables can be further refined to the subtraces whose channels are accessed. This subtrace-level precision is crucial in the soundness proof of the parallel injection axiom as it allows to drop β from $[\alpha \parallel \beta]\psi$ only if β does not write channels of ψ that are not also written by α. The signature $\Sigma(\cdot)$ of an expression denotes all occurring symbols.

Lemma 11 (Coincidence for terms and formulas). *The sets* $\mathsf{FV}(e)$ *and* $\mathsf{CN}(e)$ *are the smallest sets with the* communication-aware coincidence property for term or formula e*. That is, if* $v \downarrow \mathsf{CN}(e) = \tilde{v} \downarrow \mathsf{CN}(e)$ *on* $\mathsf{FV}(e)$ *and* $I = J$ *on* $\Sigma(e)$*, then* $Iv[\![e]\!] = J\tilde{v}[\![e]\!]$*. In particular, for formula* ϕ*:* $Iv \vDash \phi$ *iff* $J\tilde{v} \vDash \phi$*.*

Programs communicate but do *not* depend on the recorded history, thus the coincidence property for programs is not communication-aware. However, programs can produce the same communication starting from coinciding states.

Lemma 12 (Coincidence for programs). *The set* $\mathsf{FV}(\alpha)$ *is the smallest set with the* coincidence property for program α*. That is, if* $v = \tilde{v}$ *on* $X \supseteq \mathsf{FV}(\alpha)$*, and* $I = J$ *on* $\Sigma(\alpha)$*, and* $(v, \tau, w) \in I[\![\alpha]\!]$*, then* $(\tilde{v}, \tilde{\tau}, \tilde{w}) \in J[\![\alpha]\!]$ *exists such that* $w = \tilde{w}$ *on* X*, and* $\tau = \tilde{\tau}$*, and (*$w = \bot$ *iff* $\tilde{w} = \bot$*).*

3 Uniform Substitution for $\mathsf{dL}_{\mathrm{CHP}}$

In $\mathsf{dL}_{\mathrm{CHP}}$, a uniform substitution [30] σ maps function and predicate symbols to terms (of equal sort) and formulas, respectively, while substituting the arguments of the symbol for their placeholders in the replacement, and program constants are mapped to CHPs. For example, $\sigma = \{f(\cdot) \mapsto \cdot + 1, a \mapsto \mathrm{ch}(h)?v; \{x' = v\}\}$ replaces all occurrences of function symbol f with $\cdot + 1$ while the reserved 0-ary function symbol $\cdot$ marks the positions for the parameter of f in the replacement. Moreover, σ replaces the program constant a with the program $\mathrm{ch}(h)?v; \{x' = v\}$.

The key to sound uniform substitution is that new free variables must not be introduced into a context where they are bound [8]. In the presence of communication, likewise, *new channel access must not be introduced into contexts* where the channel is written (B I). For parallelism, substitution *must not reveal internals* of the parallel context to the local abstraction of a subprogram (B II), and must not violate state disjointness. The one-pass approach [32] used for $\mathsf{dL}_{\mathrm{CHP}}$ postpones these checks *and* simply applies the substitution recursively while collecting written variables and channels as taboo set (Fig. 2), thus operates linearly in the input. Clashes between the taboo, and new free variables and channel access are only checked locally at the replacement site. Likewise, clashes between the permitted channels and variables of a program constant, and its replacement program are checked locally.

The substitution operator $\sigma_Z^{U,W}(\alpha)$ for program α takes an input taboo $U \subseteq V \cup \Omega$ and a parallel context $W \subseteq V$, and returns, if defined, the substitution result and a set of output taboos $Z \subseteq V \cup \Omega$. For terms and formulas, the substitution operator σ^U only takes a taboo $U \subseteq V \cup \Omega$ as input. The substitution process clashes, i.e., prevents unsound instantiation, if it were to introduce a free variable or accessed channel into a context where it is bound (B I) *or* if it were to write variables and channels violating abstraction (B II). Moreover, substitution preserves well-formedness of programs and formulas, i.e., substitution clashes if replacements were to violate well-formedness.

$$
\begin{array}{rll}
\sigma^U(z) \equiv z & & \text{for } z \in V \\
\sigma^U(f(Y,e)) \equiv \{\cdot \mapsto \sigma^U(e \downarrow Y)\}^{\emptyset}(\sigma f(\cdot)) & & \text{if } (\mathsf{FV}(\sigma f(\cdot)) \cup \mathsf{CN}(\sigma f(\cdot))) \cap U = \emptyset \\
\sigma^U(\mathsf{op}(e_1,\dots,e_k)) \equiv \mathsf{op}(\sigma^U(e_1),\dots,\sigma^U(e_k)) & & \text{for built-in } \mathsf{op} \in \{\cdot + \cdot, \cdot \downarrow Y, \dots\} \\
\sigma^U((\theta)') \equiv (\sigma^{V \cup \Omega}(\theta))' & & \\
\hline
\sigma^U(e_1 \sim e_2) \equiv \sigma^U(e_1) \sim \sigma^U(e_2) & & \\
\sigma^U(p(Y,e)) \equiv \{\cdot \mapsto \sigma^U(e \downarrow Y)\}^{\emptyset}(\sigma p(\cdot)) & & \text{if } (\mathsf{FV}(\sigma p(\cdot)) \cup \mathsf{CN}(\sigma p(\cdot))) \cap U = \emptyset \\
\sigma^U(\neg\varphi) \equiv \neg\sigma^U(\varphi) & & \\
\sigma^U(\varphi \wedge \psi) \equiv \sigma^U(\varphi) \wedge \sigma^U(\psi) & & \\
\sigma^U(\forall z\, \varphi) \equiv \forall z\, \sigma^{U \cup \{z\}}(\varphi) & & \\
\sigma^U([\alpha]\psi) \equiv [\sigma_Z^{U,\emptyset}(\alpha)]\sigma^Z(\psi) & & \\
\sigma^U([\alpha]_{\{\mathsf{A},\mathsf{C}\}}\psi) \equiv [\sigma_Z^{U,\emptyset}(\alpha)]_{\{\sigma^Z(\mathsf{A}),\sigma^Z(\mathsf{C})\}}\sigma^Z(\psi) & & \\
\hline
\sigma_{U \cup \mathsf{BV}(\sigma a) \cup \mathsf{CN}(\sigma a)}^{U,W}(a\llparenthesis Y, \bar{z} \rrparenthesis) \equiv \sigma a & & \text{if } \mathsf{BV}(\sigma a) \subseteq \bar{z} \text{ and } \mathsf{CN}(\sigma a) = Y \\
\sigma_{U \cup \{x\}}^{U,W}(x := \theta) \equiv x := \sigma^{U \cup W}(\theta) & & \\
\sigma_{U \cup \{x\}}^{U,W}(x := *) \equiv x := * & & \\
\sigma_U^{U,W}(?\chi) \equiv\ ?\sigma^{U \cup W}(\chi) & & \\
\sigma_Z^{U,W}(\{x' = \theta \,\&\, \chi\}) \equiv \{x' = \sigma^{U \cup W}(\theta) \,\&\, \sigma^{U \cup W}(\chi)\} & & \text{with } Z = U \cup \{x, x', \mu, \mu'\} \\
\sigma_{U \cup \{\mathrm{ch}, h\}}^{U,W}(\mathrm{ch}(h)!\theta) \equiv \mathrm{ch}(h)!\sigma^{U \cup W}(\theta) & & \\
\sigma_{U \cup \{\mathrm{ch}, h, x\}}^{U,W}(\mathrm{ch}(h)?x) \equiv \mathrm{ch}(h)?x & & \\
\sigma_{Z_1 \cup Z_2}^{U,W}(\alpha \cup \beta) \equiv \sigma_{Z_1}^{U,W}(\alpha) \cup \sigma_{Z_2}^{U,W}(\beta) & & \\
\sigma_{Z_2}^{U,W}(\alpha; \beta) \equiv \sigma_{Z_1}^{U,W}(\alpha); \sigma_{Z_2}^{Z_1,W}(\beta) & & \\
\sigma_Z^{U,W}(\alpha^*) \equiv (\sigma_Z^{Z,W}(\alpha))^* & & \text{when } \sigma_Z^{U,W}(\alpha) \text{ is defined} \\
\sigma_{Z_1 \cup Z_2}^{U,W}(\alpha \parallel \beta) \equiv \sigma_{Z_1}^{U,W_{U,\beta}}(\alpha) \parallel \sigma_{Z_2}^{U,W_{U,\alpha}}(\beta) & &
\end{array}
$$

Fig. 2. Application of uniform substitution for taboo U and parallel context W, where $W_{U,\gamma} \equiv W \cup (\mathsf{BV}(\sigma^{U,W}(\gamma) \setminus (\{\mu, \mu'\} \cup V_{\mathcal{T}}))$ for any program γ, and $e \downarrow Y$ for term e is recursive push down of projection $\downarrow Y$, where $p(Y_0, e) \downarrow Y \equiv p(Y_0 \cap Y, e)$.

The side condition $(\mathsf{FV}(\sigma f(\cdot)) \cup \mathsf{CN}(\sigma f(\cdot))) \cap U = \emptyset$ implements locally that the replacement for f must not introduce free parameters that are tabooed by U (B I). The substitution $\{\cdot \mapsto \sigma^U(e \downarrow Y)\}^\emptyset$ is responsible for the argument e,[3] where $\emptyset$ suffices as the taboo U is already checked on $e \downarrow Y$. By the projection, $e{\downarrow}Y$ only depends on channels Y. Quantification $\forall z$ taboos the bound variable z. Program α in a box or ac-box has an empty parallel context $\emptyset$.

The substitution $\sigma_Z^{U,W}(\alpha)$ computes the output taboo Z by adding the written variables and channels of program α to U, e.g., real variable x for assignment $x := \theta$ and for receiving $\mathrm{ch}(h)?x$ additionally channel ch and trace variable h. The output taboo Z is passed to ac-formulas and postconditions of boxes and ac-boxes for recursive checks for clashes w.r.t. (B I). Crucially for soundness, Lemma 13 below proves that $\sigma_Z^{U,W}(\cdot)$ correctly computes the output taboo Z.

The taboo $U{\cup}W$ passed to nested expressions contains the parallel context W to prevent free variables in replacements of function and predicate symbols that are bound in parallel. This prepares the substitution process to preserve the syntax restrictions for parallel composition from previous work [6].[4] Substitution for evolution $\{x' = \theta \,\&\, \chi\}$ considers that the global time μ, μ' is always implicitly bound regardless of whether it occurs in x, x'. The fixpoint notation $\sigma_Z^{Z,W}(\alpha)$ for the replacement of repetition α^* ensures that the output taboo of the first iteration is tabooed in the subsequent iterations [32]. Computing the parallel context of α and β in case $\alpha \parallel \beta$ requires one additional pass for both subprograms because what they potentially bind after substitution adds to the parallel context of the respective other subprogram.

Lemma 13 (Correct output taboo). *Application $\sigma_Z^{U,W}(\alpha)$ of uniform substitution retains input taboo U and correctly adds the bound variables and written channels of program α, i.e., $Z \supseteq U \cup \mathsf{BV}(\sigma_Z^{U,W}(\alpha)) \cup \mathsf{CN}(\sigma_Z^{U,W}(\alpha))$.*

The side condition of $\sigma_Z^{U,W}(a(\!|Y, \bar{z}|\!))$ maintains local abstraction of subprograms (B II) because the replacement cannot bind more than $a(\!|Y, \bar{z}|\!)$, thus cannot bind variables and channels of an abstraction that is independent of $a(\!|Y, \bar{z}|\!)$. This also preserves state-disjointness (well-formedness) of parallel programs.

3.1 Semantic Effect of Uniform Substitution

The key ingredients for proving soundness of uniform substitution are Lemma 16 and 17 below. They prove that the effect of the syntactic transformation applied by uniform substitution can be equally mimicked by semantically modifying the interpretation of function and predicate symbols, and program constants. This adjoint interpretation $\sigma_w^* I$ for interpretation I and state w changes how symbols are interpreted according to their syntactic replacements in the substitution σ.

[3] Extension to vectorial arguments is straightforward.

[4] For $\alpha \parallel \beta$, the restriction is $(\mathsf{V}(\alpha) \cap \mathsf{BV}(\beta)) \cup (\mathsf{V}(\beta) \cap \mathsf{BV}(\alpha)) \subseteq \{\mu, \mu'\} \cup V_{\mathcal{T}}$ [6]. However, in this paper, programs obey a less restrictive syntax for simplicity.

Definition 14 (Adjoint substitution). *For interpretation I and state w, the* adjoint interpretation $\sigma_w^* I$ *changes the meaning of function and predicate symbols, and program constants according to the substitution σ evaluated in state w:*

$$\begin{aligned}
&\sigma_w^* I(f^{\mathbb{M}} : \mathbb{M}_{\text{arg}}) : \mathbb{M}_{\text{arg}} \to \mathbb{M}; d \mapsto I_\cdot^d w[\![\sigma f(\cdot)]\!] && \textit{where } \mathbb{M}, \mathbb{M}_{\text{arg}} \in \{\mathbb{R}, \mathbb{N}, \Omega, \mathcal{T}\}\\
&\sigma_w^* I(p : \mathbb{M}_{\text{arg}}) = \{d \in \mathbb{M}_{\text{arg}} \mid I_\cdot^d w \vDash \sigma p(\cdot)\} && \textit{where } \mathbb{M}_{\text{arg}} \in \{\mathbb{R}, \mathbb{N}, \Omega, \mathcal{T}\}\\
&\sigma_w^* I(a(\!|Y, \bar{z}|\!)) = I[\![\sigma a]\!]
\end{aligned}$$

We follow the observation for dGL [32] that the more liberal one-pass substitution requires stronger coincidence between the substitution and the adjoint on neighborhoods of the original state. Where the dGL soundness proof has succeeded by a neighborhood semantics of state on taboos, the dL_{CHP} proof succeeds with a generalization to a neighborhood semantics of state and communication on taboos. The neighborhood of a state consists of its variations:

Definition 15 (Variation). *For a set $U \subseteq V \cup \Omega$, a state v is a U*-variation of state *w if v and w only differ on variables or projections onto channels in U, i.e., $v \downarrow (U^{\complement} \cap \Omega) = w \downarrow (U^{\complement} \cap \Omega)$ on $U^{\complement} \cap V$.*

The proofs of Lemma 16 and 17 follow a lexicographic induction on the structure of substitution, and term, formula, or program. In Lemma 17, the induction is mutual for formulas and programs.

Lemma 16 (Semantic uniform substitution). *The term e evaluates equally over U-variations under uniform substitution σ^U and adjoint interpretation $\sigma_w^* I$, i.e., $Iv[\![\sigma^U(e)]\!] = \sigma_w^* Iv[\![e]\!]$ for all U-variations v of w.*

Lemma 17 (Semantic uniform substitution). *The formula ϕ and the program α have equal truth value and semantics, respectively, over U-variations under uniform substitution σ^U and adjoint interpretation $\sigma_w^* I$, i.e.,*

1. *for all U-variations v of w: $Iv \vDash \sigma^U(\phi)$ iff $\sigma_w^* Iv \vDash \phi$*
2. *for all $(U \cup W)$-variations v of w: $(v, \tau, o) \in I[\![\sigma_Z^{U,W}(\alpha)]\!]$ iff $(v, \tau, o) \in \sigma_w^* I[\![\alpha]\!]$*

3.2 Uniform Substitution Proof Rule

The proof rule US for uniform substitution is the single point of truth for the sound instantiation of axioms (plus renaming of bound variables [30] and written channels, e.g., $[x := \theta]\psi(x)$ to $[y := \theta]\psi(y)$ and $[\text{ch}(h)?x]\psi(\text{ch})$ to $[\text{dh}(h)?x]\psi(\text{dh})$. Soundness of the rule, i.e., that validity of its premise implies validity of the conclusion, immediately follows from Lemma 17. Since the substitution process starts with no taboos, $\sigma(\phi)$ is short for $\sigma^{\emptyset}(\phi)$. If the substitution clashes, i.e., $\sigma^{\emptyset}(\phi)$ is not defined, then rule US is not applicable.

Theorem 18 (US is sound). *The proof rule US is sound.*

$$\frac{\phi}{\sigma(\phi)}\ \text{US}$$

Unlike dL [30] and dGL [32], dL_{CHP} has a context-sensitive syntax for programs and formulas (see Definition 2 and Definition 4). By Proposition 19, uniform substitution, however, preserves syntactic well-formedness. Since all axioms in Sect. 4 will be well-formed, only well-formed formulas can be derived in dL_{CHP}.

Proposition 19 (US preserves well-formedness). *The result $\sigma^U(\phi)$ (if defined) of applying uniform substitution to a well-formed formula ϕ is well-formed.*

4 Axiomatic Proof Calculus

Figure 3 presents a sound proof calculus for dL_{CHP}. The significant difference to dL_{CHP}'s schematic calculus [6] is that it completely abandons soundness-critical side conditions, internalizing them syntactically in the axioms. Only axiom $[]_{\mathrm{W}\mathsf{A}}$ was adjusted to obtain a symbolic representation and an ac-version K_{AC} of modal modus ponens is included. Now, distribution of ac-boxes over conjuncts $[]_{\mathsf{AC}}\wedge$ and ac-monotonicity $\mathrm{M}[\cdot]_{\mathsf{AC}}$ derive from K_{AC}, thus are dropped. Except for the small changes soundness is inherited from the schematic axioms [6].

Algebraic laws for reasoning about traces [6] can be easily adapted to uniform substitution as well [7]. Decidable first-order real arithmetic [41] and Presburger arithmetic [34] have corresponding oracle proof rules [6].

Remark 20. To obtain a truly finite list of axioms from Fig. 3, symbolic (co)finite sets can be finitely axiomatized as a boolean algebra together with extensionality, which can be unrolled to a finite disjunction for (co)finite sets [7].

Parallel Composition. The parallel injection axiom $[\|\_]_{\mathsf{AC}}$ in Fig. 3 decomposes parallel CHPs by local abstraction (B II). Unlike dL_{CHP}'s [6] and Hoare-style [46,47] schematic calculi for ac-reasoning, axiom $[\|\_]_{\mathsf{AC}}$ internalizes the noninterference property [6, Def. 7] that determines valid instances of formula

$$[\alpha]_{\{\mathsf{A},\mathsf{C}\}}\psi \to [\alpha \parallel \beta]_{\{\mathsf{A},\mathsf{C}\}}\psi \tag{1}$$

purely syntactically. To focus on noninterference, $a\llparenthesis Y_a, \bar{z}_a \rrparenthesis \parallel_{\mathrm{wf}} b\llparenthesis Y_b, \bar{z}_b \rrparenthesis$ abbreviates well-formed parallel composition $a\llparenthesis Y_a, \bar{z}_a \rrparenthesis \parallel b\llparenthesis Y_b, (\bar{z}_b \cap \bar{z}_a^{\complement}) \cup \{\mu, \mu'\} \cup V_{\mathcal{T}} \rrparenthesis$ using operator $\parallel_{\mathrm{wf}}$ for program constants $a\llparenthesis Y_a, \bar{z}_a \rrparenthesis$, $b\llparenthesis Y_b, \bar{z}_b \rrparenthesis$. This notation ensures disjoint parallel state except for the global time μ, μ' and recorder variables $V_{\mathcal{T}}$.

Intuitively, axiom $[\|\_]_{\mathsf{AC}}$ restricts β in Eq. (1) such that α overapproximates the behavior of $\alpha \parallel \beta$ influencing A, C, or ψ. For this purpose, noninterference internalized in $b\llparenthesis Y_b \cap (Y^{\complement} \cup Y_a), \bar{z}^{\complement} \rrparenthesis$ forbids b to bind variables $\bar{z}$ that are free in the postcondition $p(Y, \bar{z})$, and $Y^{\complement}$ forbids b to bind channels Y (except for channels Y_a written by a because joint parallel communication can already be observed from a, too). Moreover, parallel programs always agree on the global time μ, μ' and the communication recorded by trace variables $V_{\mathcal{T}}$. Therefore, the operator $\parallel_{\mathrm{wf}}$ explicitly allows their sharing even if $\bar{z}^{\complement}$ disallows it. Note that Y_a and Y, and $\bar{z}_a$ and $\bar{z}$ may overlap but can also be disjoint.

$$[{:=}]\quad [x := g^{\mathbb{R}}]p(x) \leftrightarrow p(g^{\mathbb{R}})^{a}$$

$$[;]_{\mathsf{AC}}\quad [a;b]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P} \leftrightarrow [a]_{\{\mathrm{R},\mathrm{Q}\}}[b]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P}$$

$$[{:*}]\quad [x := *]p(x) \leftrightarrow \forall x\, p(x)$$

$$[\cup]_{\mathsf{AC}}\quad [a \cup b]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P} \leftrightarrow [a]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P} \wedge [b]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P}$$

$$[?]\quad [?q_{\mathbb{R}}]p \leftrightarrow (q_{\mathbb{R}} \rightarrow p)^{a}$$

$$[{}^{*}]_{\mathsf{AC}}\quad [a^{*}]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P} \leftrightarrow [a^{0}]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P} \wedge [a]_{\{\mathrm{R},\mathrm{Q}\}}[a^{*}]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P}^{b}$$

$$[]_{\top,\top}\quad [a]\mathrm{P} \leftrightarrow [a]_{\{\top,\top\}}\mathrm{P}$$

$$[]\mathrm{W}\mathsf{A}\quad [a]_{\{\top,\mathrm{W}_{\mathsf{A}}\}}\top \wedge [a]_{\{\mathrm{R}_1\wedge\mathrm{R}_2,\mathrm{Q}_1\wedge\mathrm{Q}_2\}}\mathrm{P} \rightarrow [a]_{\{\mathrm{R},\mathrm{Q}_1\wedge\mathrm{Q}_2\}}\mathrm{P}^{c}$$

$$[\|\_]_{\mathsf{AC}}\quad [a\llparenthesis Y_a, \bar{z}_a\rrparenthesis]_{\{\mathrm{R},\mathrm{Q}\}}p(Y,\bar{z}) \rightarrow [a\llparenthesis Y_a, \bar{z}_a\rrparenthesis \;\|_{\mathrm{wf}}\; b\llparenthesis Y_b \cap (Y^{\complement} \cup Y_a), \bar{z}^{\complement}\rrparenthesis]_{\{\mathrm{R},\mathrm{Q}\}}p(Y,\bar{z})^{d}$$

$$[\mu]\quad [\{\bar{x}' = g^{\mathbb{R}}(\bar{x},\mu) \;\&\; q_{\mathbb{R}}(\bar{x},\mu)\}]p(\bar{x},\mu) \leftrightarrow [\{\mu' = 1, \bar{x}' = g^{\mathbb{R}}(\bar{x},\mu) \;\&\; q_{\mathbb{R}}(\bar{x},\mu)\}]p(\bar{x},\mu)^{a}$$

$$[\mathrm{ch}!]\quad [\mathrm{ch}(h)!g^{\mathbb{R}}]p(\mathrm{ch},h) \leftrightarrow \forall h_0\, \big(h_0 = h \cdot \langle \mathrm{ch}, g^{\mathbb{R}}, \mu\rangle \rightarrow p(\mathrm{ch}, h_0)\big)$$

$$[\mathrm{ch}!]_{\mathsf{AC}}\quad [\mathrm{ch}(h)!g^{\mathbb{R}}]_{\{\widehat{r},\widehat{q}\}}\widehat{p} \leftrightarrow \widehat{q} \wedge \Big(\widehat{r} \rightarrow [\mathrm{ch}(h)!g^{\mathbb{R}}]\big(\widehat{q} \wedge (\widehat{r} \rightarrow \widehat{p})\big)\Big)$$

$$[\mathrm{ch}?]_{\mathsf{AC}}\quad [\mathrm{ch}(h)?x]_{\{\widehat{r},\widehat{q}\}}p(\mathrm{ch},h,x) \leftrightarrow [x := *][\mathrm{ch}(h)!x]_{\{\widehat{r},\widehat{q}\}}p(\mathrm{ch},h,x)$$

$$[\epsilon]_{\mathsf{AC}}\quad [a\llparenthesis \emptyset, V_{\mathbb{R}}\rrparenthesis]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P} \leftrightarrow \mathrm{Q} \wedge (\mathrm{R} \rightarrow [a\llparenthesis \emptyset, V_{\mathbb{R}}\rrparenthesis]\mathrm{P})$$

$$\mathrm{W}[]_{\mathsf{AC}}\quad [a]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P} \leftrightarrow \mathrm{Q} \wedge [a]_{\{\mathrm{R},\mathrm{Q}\}}(\mathrm{Q} \wedge (\mathrm{R} \rightarrow \mathrm{P}))$$

$$\mathrm{I}_{\mathsf{AC}}\quad [a^{*}]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P} \leftrightarrow [a^{0}]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P} \wedge [a^{*}]_{\{\mathrm{R},\top\}}(\mathrm{P} \rightarrow [a]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P})$$

$$\mathrm{K}_{\mathsf{AC}}\quad [a]_{\{\mathrm{R},\mathrm{Q}_1\rightarrow\mathrm{Q}_2\}}(\mathrm{P}_1 \rightarrow \mathrm{P}_2) \rightarrow ([a]_{\{\mathrm{R},\mathrm{Q}_1\}}\mathrm{P}_1 \rightarrow [a]_{\{\mathrm{R},\mathrm{Q}_2\}}\mathrm{P}_2)$$

$$\mathrm{MP}\ \frac{p \rightarrow q \quad p}{q}$$

$$\mathrm{G}_{\mathsf{AC}}\ \frac{\mathrm{Q} \wedge \mathrm{P}}{[a]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P}}$$

$$\forall\ \frac{p(x)}{\forall x\, p(x)}$$

$$\mathrm{CE}\ \frac{\mathrm{P}_1 \leftrightarrow \mathrm{P}_2}{C(\mathrm{P}_1) \leftrightarrow C(\mathrm{P}_2)}$$

$\mathrm{P}_j \equiv p_j(Y,\bar{z})$, and $\mathrm{R}_j \equiv r_j(Y,\bar{h})$, and $\mathrm{Q}_{\underline{j}} \equiv q_j(Y,\bar{h})$, and $\widehat{\chi} \equiv \chi(\mathrm{ch},h)$, where j may be blank, and $Y \subseteq \mathit{\Omega}$, $\bar{z} \subseteq V_{\mathbb{R}} \cup V_{\mathcal{T}}$, and $\bar{h} \subseteq V_{\mathcal{T}}$ are (co)finite.

[a] Replacements for function symbol $g^{\mathbb{R}}$ and predicate symbol $q_{\mathbb{R}}$ are restricted to polynomials in $V_{\mathbb{R}}$ and first-order real arithmetic, respectively.

[b] Recall that $[\alpha^{0}]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P} \leftrightarrow \mathrm{Q} \wedge (\mathrm{R} \rightarrow \mathrm{P})$ by $[\epsilon]_{\mathsf{AC}}$ and $[?]$ since $\alpha^{0} \equiv\ ?\top$.

[c] W_{A} is the compositionality condition $(\mathrm{R} \wedge \mathrm{Q}_1 \rightarrow \mathrm{R}_2) \wedge (\mathrm{R} \wedge \mathrm{Q}_2 \rightarrow \mathrm{R}_1)$.

[d] The operator $\|_{\mathrm{wf}}$ abbreviates well-formed parallel composition (see above).

Fig. 3. $\mathsf{dL}_{\mathrm{CHP}}$ proof calculus

Despite its asymmetric shape, axiom $[\|\_]_{\mathsf{AC}}$ decomposes $[\alpha\|\beta](\phi\wedge\psi)$ into $[\alpha]\phi$ and $[\beta]\psi$ (if they mutually do not interfere) via independent proofs for $[\alpha\|\beta]\phi$ and $[\alpha\|\beta]\psi$, which drop either α or β by $[\|\_]_{\mathsf{AC}}$ modulo commutativity.

Axiom System. For each program statement, there is either a dynamic or an ac-axiom because the respective other version derives by axiom $[]_{\top,\top}$ or $[\epsilon]_{\mathsf{AC}}$. Axioms $[{:=}]$, $[{:*}]$, and $[?]$ are as in dL [30]. Axioms $[;]_{\mathsf{AC}}$, $[\cup]_{\mathsf{AC}}$, and $[{}^{*}]_{\mathsf{AC}}$ for decomposition, and I_{AC} for induction carefully generalize their versions in differential [30] dynamic [14] logic to ac-reasoning. Sending is handled step-wise via flattening the assumption-commitments by axiom $[\mathrm{ch}!]_{\mathsf{AC}}$ and axiom $[\mathrm{ch}!]$ that executes the effect onto the recorder h. The duality $[\mathrm{ch}?]_{\mathsf{AC}}$ turns receiving into arbitrary sending, which only synchronizes if it agrees with the parallel context on the value. Usage of axiom $\mathrm{W}[]_{\mathsf{AC}}$ is for convenience. Axiom $[\mu]$ materializes the flow of global time μ such that dL's axiomatization of continuous evolution [30] gets applicable, which requires ODE shape $\bar{x}' = f^{\mathbb{R}}(\bar{x})$. The axiomatic proof rules G_{AC}, MP, $\forall$, and CE are an ac-version of Gödels generalization rule, modus ponens, quantifier elimination, and contextual equivalence, respectively.

The axiom $[]_{\text{W}\mathsf{A}}$ can weaken assumptions. Its slight change compared to dL_{CHP}'s schematic calculus [6] exploits that the compositionality condition W_{A} is only required for a's reachable worlds. Interestingly, dL_{CHP}'s monotonicity rule $\text{M}[\cdot]_{\mathsf{AC}}$ [6] does not derive from modal modus ponens K_{AC} and Gödel generalization G_{AC} in analogy to dL [30] but needs $\text{W}[]_{\mathsf{AC}}$ handling monotonicity of assumptions, which does not fit into G_{AC} because necessitating the assumption in G_{AC} would render the derivation of $[\alpha]_{\{\bot,\top\}}\top$ by G_{AC} impossible.

Axioms using postcondition $\text{P} \equiv p(Y, \bar{z})$, e.g., in $[;]_{\mathsf{AC}}$, allow any replacement of P since accessed channels $Y \subseteq \Omega$ and free variables $\bar{z} \subseteq V_{\mathbb{R}} \cup V_{\mathcal{T}}$ can be arbitrary. Replacements of assumptions $\text{R} \equiv r(Y, \bar{h})$ and commitments $\text{Q} \equiv q(Y, \bar{h})$ can instead only mention trace variables $\bar{h} \subseteq V_{\mathcal{T}}$ bound in their context. This reflects that trace variables are the only interface between the program α and the ac-formulas A and C in an ac-box $[\alpha]_{\{\mathsf{A},\mathsf{C}\}}\psi$ (well-formedness).

Theorem 21 (Soundness). *The proof calculus for dL_{CHP} presented in Fig. 3 is sound as an instantiation of the schematic calculus [6].*

Clashes. Clashes sort out unsound instantiations of axioms. Unlike in dL and dGL [30,32] whose clashes are solely due to tabooed variables in terms and formulas, clashes in dL_{CHP} can also be due to tabooed channels, and even due to taboos in programs. For example, the substitution $\sigma = \{a \mapsto \text{gh}(h)!1, b \mapsto \text{ch}(h)!2, p \mapsto \psi, r \mapsto \top, q \mapsto \top\}$ with $\psi \equiv |h \downarrow \text{ch}| > 0 \wedge |h \downarrow \text{dh}| > 0 \wedge y < 0$ clashes below, where $Y = \{\text{ch}, \text{dh}\}$, and $\bar{z} \equiv h, y$, and $\text{R} \equiv r(Y)$, and $\text{Q} \equiv q(Y)$. Writing channel ch in the replacement for b would break the local abstraction of a as ch is accessed in ψ but not written in the replacement for a, thus the clash indeed sorts out an unsound instantiation.

$$\dfrac{[a\llparenthesis\{\text{gh}\}, h\rrparenthesis]_{\{\text{R},\text{Q}\}}p(Y, \bar{z}) \rightarrow [a\llparenthesis\{\text{gh}\}, h\rrparenthesis \parallel_{\text{wf}} b\llparenthesis\{\text{ch}\}\cap(Y^{\mathsf{C}}\cup\{\text{gh})\}, \bar{z}^{\mathsf{C}}\rrparenthesis]_{\{\text{R},\text{Q}\}}p(Y, \bar{z})}{[\text{gh}(h)!1]_{\{\top,\top\}}\psi \rightarrow [\text{gh}(h)!1 \parallel \text{ch}(h)!2]_{\{\top,\top\}}\psi}\ \lightning\,\text{clash}$$

In contrast, $\sigma = \{a \mapsto \text{ch}(h)?x; \text{gh}(h)!1, b \mapsto \text{ch}(h)!2, p \mapsto \psi, r \mapsto \top, q \mapsto \top\}$ does not clash below, where $Y = \{\text{ch}, \text{dh}\}$, and $Y_a = \{\text{ch}, \text{gh}\}$, and other abbreviations are as above, because $\text{ch} \in Y^{\mathsf{C}} \cup Y_a = \{\text{dh}\}^{\mathsf{C}}$. Intuitively, the ch-communication of b remains observable after dropping b from the parallel composition as it is joint with a.

$$\dfrac{\dfrac{*}{[a\llparenthesis Y_a, h, x\rrparenthesis]_{\{\text{R},\text{Q}\}}p(Y, \bar{z}) \rightarrow [a\llparenthesis Y_a, h, x\rrparenthesis \parallel_{\text{wf}} b\llparenthesis\{\text{ch}\} \cap (Y^{\mathsf{C}} \cup Y_a), \bar{z}^{\mathsf{C}}\rrparenthesis]_{\{\text{R},\text{Q}\}}p(Y, \bar{z})}\ [\|\_]_{\mathsf{AC}}}{[\text{ch}(h)?x; \text{gh}(h)!1]_{\{\top,\top\}}\psi \rightarrow [(\text{ch}(h)?x; \text{gh}(h)!1) \parallel \text{ch}(h)!2]_{\{\top,\top\}}\psi}\ \text{US}$$

Also note that by the operator $\parallel_{\text{wf}}$ for well-formed parallel composition, the recorder variable h can be shared without causing a clash above. However, clashes prevent instantiation that would violate syntactic well-formedness of programs (Definition 2) by binding the same state variable in parallel:

$$\dfrac{[a(\!|\emptyset, x|\!)]_{\{r,q\}}p(x,y) \to [a(\!|\emptyset, x|\!) \parallel_{\mathrm{wf}} b(\!|\emptyset, \{x,y\}^{\complement}|\!)]_{\{r,q\}}p(x,y)}{[x := y]_{\{\mathsf{T},\mathsf{T}\}}y = x \to [x := y \parallel x := 0]_{\{\mathsf{T},\mathsf{T}\}}y = x}\;\text{↯clash}$$

Well-formedness of programs and formulas is ensured in the axioms by well-formed parallel composition $\|_{\mathrm{wf}}$ and limitation to trace variables $\bar{h}$ in $\mathrm{R}_j \equiv r_j(Y, \bar{h})$ and $\mathrm{Q}_j \equiv q_j(Y, \bar{h})$ in ac-boxes $[\alpha]_{\{\mathrm{R}_j,\mathrm{Q}_j\}}\psi$ in Fig. 3, respectively. By Proposition 19, uniform substitution always preserves well-formedness.

Example 22. The proof tree below decomposes safety (Example 5) of cruise control (Example 3) into safety ① of controller ct and branch ② to be continued to safety of the vehicle ve. The lower subproof introduces the ac-formulas

$$\mathsf{A} \equiv \mathsf{C} \equiv \big(|h \downarrow \mathrm{tar}| > 0 \to 0 \leq \mathtt{val}(h \downarrow \mathrm{tar}) \leq V\big)$$

using axiom $[]_{\mathrm{W}\mathsf{A}}$ to abstract from the communication between ct and ve. The upper subproof uses the parallel injection axiom $[\|\_]_{\mathsf{AC}}$ to drop ve. Uniform substitution US does not clash as the commitment C only refers to joint communication of ct and ve. Other applications of US (e.g., for $[]_{\mathrm{W}\mathsf{A}}$) are omitted. Rule Prop denotes propositional reasoning. Abbreviations are as follows: $\alpha \equiv a(\!|\mathrm{tar}, v^{\mathrm{tr}}_{\mathsf{ct}}, t, t', \mu, \mu', h|\!)$, $\mathrm{R} \equiv r(\mathrm{tar}, h)$, $\mathrm{Q} \equiv q(\mathrm{tar}, h)$, $\mathrm{P} \equiv p(\mathrm{tar})$.

$$\dfrac{\dfrac{\dfrac{\textcircled{1}}{\varphi \to [\mathsf{ct}^*]_{\{\mathsf{T},\mathsf{C}\}}\mathsf{T}} \qquad \dfrac{\dfrac{*}{[\alpha]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P} \to [\alpha \parallel_{\mathrm{wf}} b(\!|\mathrm{tar}, v^{\mathrm{tr}}_{\mathrm{ve}}, a_{\mathrm{ve}}, t_0, v_{\mathrm{ve}}, v'_{\mathrm{ve}}|\!)]_{\{\mathrm{R},\mathrm{Q}\}}\mathrm{P}}\;[\|\_]_{\mathsf{AC}}}{[\mathsf{ct}^*]_{\{\mathsf{T},\mathsf{C}\}}\mathsf{T} \to [\mathsf{ct}^* \| \mathsf{ve}^*]_{\{\mathsf{T},\mathsf{C}\}}\mathsf{T}}\;\mathrm{US}}{\varphi \to [\mathsf{ct}^* \| \mathsf{ve}^*]_{\{\mathsf{T},\mathsf{C}\}}\mathsf{T}}\;\mathrm{MP,\ CE}}{\varphi \to [\mathsf{ct}^* \| \mathsf{ve}^*]_{\{\mathsf{T}\wedge\mathsf{A},\mathsf{C}\}}\mathsf{T}}\;\mathrm{M}[\cdot]_{\mathsf{AC}}$$

$$\dfrac{\dfrac{\dfrac{\mathrm{G}_{\mathsf{AC}}\;\dfrac{\mathrm{Prop}\;\dfrac{*}{(\mathsf{C} \to \mathsf{A}) \wedge \mathsf{T}}}{\varphi \to [\mathsf{ct}^* \| \mathsf{ve}^*]_{\{\mathsf{T},\mathsf{C}\to\mathsf{A}\}}\mathsf{T}} \qquad \dfrac{\dfrac{\dfrac{\textcircled{2}}{\varphi \to [\mathsf{ct}^* \| \mathsf{ve}^*]_{\{\mathsf{T}\wedge\mathsf{A},\mathsf{T}\}}\psi_{\mathrm{safe}}}}{\varphi \to [\mathsf{ct}^* \| \mathsf{ve}^*]_{\{\mathsf{T}\wedge\mathsf{A},\mathsf{C}\}}\mathsf{T} \wedge [\mathsf{ct}^* \| \mathsf{ve}^*]_{\{\mathsf{T}\wedge\mathsf{A},\mathsf{T}\}}\psi_{\mathrm{safe}}}\;\wedge\mathrm{R}}{\varphi \to [\mathsf{ct}^* \| \mathsf{ve}^*]_{\{\mathsf{T}\wedge\mathsf{A},\mathsf{C}\wedge\mathsf{T}\}}(\mathsf{T} \wedge \psi_{\mathrm{safe}})}\;[]_{\mathsf{AC}}\wedge}{\varphi \to [\mathsf{ct}^* \| \mathsf{ve}^*]_{\{\mathsf{T},\mathsf{C}\to\mathsf{A}\}}\mathsf{T} \wedge [\mathsf{ct}^* \| \mathsf{ve}^*]_{\{\mathsf{T}\wedge\mathsf{A},\mathsf{C}\wedge\mathsf{T}\}}(\mathsf{T} \wedge \psi_{\mathrm{safe}})}\;\wedge\mathrm{R}}{\varphi \to [\mathsf{ct}^* \| \mathsf{ve}^*]_{\{\mathsf{T},\mathsf{C}\wedge\mathsf{T}\}}(\mathsf{T} \wedge \psi_{\mathrm{safe}})}\;[]_{\mathrm{W}\mathsf{A}}}{\varphi \to [\mathsf{ct}^* \| \mathsf{ve}^*]\psi_{\mathrm{safe}}}\;[]_{\top,\top},\ \mathrm{M}[\cdot]_{\mathsf{AC}}$$

5 Related Work

Uniform substitution for differential dynamic logic dL [30] generalizes Church's uniform substitution for first-order logic [8, §35, 40]. Unlike the lifting from dL to differential game logic dGL [31], $\mathsf{dL}_{\mathrm{CHP}}$ generalizes into the complementary direction of communication and parallelism. Unlike schematic calculi [2,19,27,44,46], whose treacherous schematic simplicity relies on encoding all subtlety of parallel systems in significant soundness-critical side conditions, our development builds upon a minimalistic non-schematic parallel injection axiom *and* sound instantiation encapsulated in uniform substitution. This provides a new, more atomic and more modular understanding of parallel systems overcoming the root cause for large soundness-critical prover kernels [5,9,12,16,18,36]. Usage of uniform substitution reduced the kernel of the theorem prover KeYmaera from 105 kLOC to 2 kLOC in KeYmaera X [23]. We expect dL_{CHP}'s integration into KeYmaera X to stay in the same order of magnitude.

To the best of our knowledge, assumption-commitment reasoning [22,46][5] has no tool support, which might be due to vast implementation effort. The latter can be underpinned by analogy with tools [5,9,16,18,36] for verification of shared-variables concurrency, some of which use rely-guarantee reasoning [36,39]. Unlike uniform substitution for dL_{CHP} that enables a straightforward implementation of a small prover kernel, they all rely on large soundness-critical code bases. Unlike refinement checking for CSP [12] and discrete-time CSP [4], dL_{CHP} supports safety properties of dense-time hybrid systems. Contrary to our goal of small prover kernels, implementations of model checkers [12] are inherently large.

Beyond embeddings of concurrency reasoning for discrete systems into proof assistants [3,25,26,38], dL_{CHP} can verify parallel hybrid systems synchronizing in shared global time. The latter imposes even more complicated binding structures than parallel or hybrid systems alone but dL_{CHP}'s uniform substitution calculus continues to manage them in a modular way.

The recent tool HHLPy [37] for hybrid CSP (HCSP) [17] is limited to the sequential fragment. Unlike extending HHLPy to parallelism, which would require extensive soundness-critical side conditions and a treatment of the duration calculus, integrating dL_{CHP} into KeYmaera X [11] boils down to adding a finite list of concrete object level formulas as axioms and only small changes to the uniform substitution process. In contrast to dL_{CHP}'s compositional parallel systems calculus [6], HCSP calculi [13,20,42] are non-compositional [6] as they either unroll exponentially many interleavings from the operational semantics [13,42] or can only decompose independent parallel components [20] causing limited ability to reason about complex systems. Former HCSP tools [43,45] only implement a non-compositional calculus [20] reinforcing the significance of our approach for managing parallel hybrid systems reasoning. Other hybrid process algebras defer to model checkers for reasoning [10,21,40]. Further discussion of dL_{CHP} is in [6].

6 Conclusion

This paper introduced a sound one-pass uniform substitution calculus for the dynamic logic of communicating hybrid programs dL_{CHP} thereby mastering the significant challenge of developing simple sound proof calculi for parallel hybrid systems with communication. Uniform substitution can separate even notoriously complicated binding structures from parallelism with communication in multi-dynamical logics into axioms and their instantiation. In the case of dL_{CHP},

[5] Assumption-commitment and rely-guarantee reasoning are specific patterns for message-passing and shared variables concurrency, respectively. The broader assume-guarantee principle has been used across diverse areas for various purposes.

this applies to channel access in predicates and the need for local abstraction of subprograms in parallel statements, and it even turns out that uniform substitution can maintain a context-sensitive syntax along the way. Thanks to uniform substitution, parallel systems reasoning reduces to multiple uses of an asymmetric parallel injection axiom.

Now, with uniform substitution a straightforward implementation of $\mathsf{dL}_{\mathrm{CHP}}$ in KeYmaera X is only one step away.

Acknowledgments. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY), an Alexander von Humboldt Professorship, and by the AFOSR under grant number FA9550-16-1-0288.

References

1. Apt, K.R., de Boer, F.S., Olderog, E.R.: Verification of Sequential and Concurrent Programs, 3rd edn. Springer, Heidelberg (2010). https://doi.org/10.1007/978-1-84882-745-5
2. Apt, K.R., Francez, N., de Roever, W.P.: A proof system for communicating sequential processes. ACM Trans. Program. Lang. Syst. **2**(3), 359–385 (1980). https://doi.org/10.1145/357103.357110
3. Armstrong, A., Gomes, V.B.F., Struth, G.: Algebras for program correctness in Isabelle/HOL. In: Höfner, P., Jipsen, P., Kahl, W., Müller, M.E. (eds.) RAMICS 2014. LNCS, vol. 8428, pp. 49–64. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06251-8_4
4. Armstrong, P.J., Lowe, G., Ouaknine, J., Roscoe, B.: Model checking timed CSP. In: Voronkov, A., Korovina, M.V. (eds.) HOWARD-60: A Festschrift on the Occasion of Howard Barringer's 60th Birthday, EPiC Series in Computing, vol. 42, pp. 13–33. EasyChair (2014). https://doi.org/10.29007/6fqk
5. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 102–110. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_7
6. Brieger, M., Mitsch, S., Platzer, A.: Dynamic logic of communicating hybrid programs. CoRR abs/2302.14546 (2023). https://doi.org/10.48550/arXiv.2302.14546
7. Brieger, M., Mitsch, S., Platzer, A.: Uniform substitution for dynamic logic with communicating hybrid programs. CoRR abs/2303.17333 (2023). https://doi.org/10.48550/arXiv.2303.17333
8. Church, A.: Introduction to Mathematical Logic. Princeton University Press, Princeton (1956)
9. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_2
10. Cong, X., Yu, H., Xu, X.: Verification of hybrid chi model for cyber-physical systems using PHAVer. In: Barolli, L., You, I., Xhafa, F., Leu, F., Chen, H. (eds.) Proceedings of the 7th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), pp. 122–128. IEEE Computer Society (2013). https://doi.org/10.1109/IMIS.2013.29

11. Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 527–538. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_36
12. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3—a modern refinement checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 187–201. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_13
13. Guelev, D.P., Wang, S., Zhan, N.: Compositional Hoare-style reasoning about hybrid CSP in the duration calculus. In: Larsen, K.G., Sokolsky, O., Wang, J. (eds.) SETTA 2017. LNCS, vol. 10606, pp. 110–127. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69483-2_7
14. Harel, D. (ed.): First-Order Dynamic Logic. LNCS, vol. 68. Springer, Heidelberg (1979). https://doi.org/10.1007/3-540-09237-4
15. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978). https://doi.org/10.1145/359576.359585
16. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
17. Jifeng, H.: From CSP to hybrid systems In: A Classical Mind: Essays in Honour of C. A. R. Hoare, pp. 171–189. Prentice Hall International (1994)
18. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
19. Levin, G., Gries, D.: A proof technique for communicating sequential processes. Acta Informatica **15**(3), 281–302 (1981). https://doi.org/10.1007/BF00289266
20. Liu, J., et al.: A calculus for hybrid CSP. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 1–15. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17164-2_1
21. Man, K.L., Reniers, M.A., Cuijpers, P.J.L.: Case studies in the hybrid process algebra HyPA. Int. J. Softw. Eng. Knowl. Eng. **15**(2), 299–306 (2005). https://doi.org/10.1142/S0218194005002385
22. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE Trans. Softw. Eng. **7**(4), 417–426 (1981). https://doi.org/10.1109/TSE.1981.230844
23. Mitsch, S., Platzer, A.: A retrospective on developing hybrid system provers in the KeYmaera family. In: Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Ulbrich, M. (eds.) Deductive Software Verification: Future Perspectives. LNCS, vol. 12345, pp. 21–64. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64354-6_2
24. Müller, A., Mitsch, S., Retschitzegger, W., Schwinger, W., Platzer, A.: A component-based approach to hybrid systems safety verification. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 441–456. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_28
25. Nieto, L.P.: Verification of parallel programs with the Owicki-Gries and Rely-Guarantee methods in Isabelle/HOL. Ph.D. thesis, Technical University Munich, Germany (2002). http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/prensa_nieto.html
26. Nipkow, T., Nieto, L.P.: Owicki/Gries in Isabelle/HOL. In: Finance, J.-P. (ed.) FASE 1999. LNCS, vol. 1577, pp. 188–203. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-540-49020-3_13

27. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**, 319–340 (1976). https://doi.org/10.1007/BF00268134
28. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reas. **41**(2), 143–189 (2008). https://doi.org/10.1007/s10817-008-9103-8
29. Platzer, A.: Differential game logic. ACM Trans. Comput. Log. **17**(1), 1:1–1:51 (2015). https://doi.org/10.1145/2817824
30. Platzer, A.: A complete uniform substitution calculus for differential dynamic logic. J. Autom. Reas. **59**(2), 219–265 (2017). https://doi.org/10.1007/s10817-016-9385-1
31. Platzer, A.: Uniform substitution for differential game logic. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 211–227. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_15
32. Platzer, A.: Uniform substitution at one fell swoop. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 425–441. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_25
33. Platzer, A., Tan, Y.K.: Differential equation invariance axiomatization. J. ACM **67**(1), 6:1–6:66 (2020). https://doi.org/10.1145/3380825
34. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Comptes Rendus du I congrès de Mathématiciens des Pays Slaves (1931)
35. de Roever, W.P., et al.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press (2001)
36. Schellhorn, G., Bodenmüller, S., Bitterlich, M., Reif, W.: Software & system verification with KIV. In: Ahrendt, W., Beckert, B., Bubel, R., Johnsen, E.B. (eds.) The Logic of Software. A Tasting Menu of Formal Methods. LNCS, vol. 13360, pp. 408–436. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-08166-8_20
37. Sheng, H., Bentkamp, A., Zhan, B.: HHLPy: practical verification of hybrid systems using Hoare logic. In: Chechik, M., Katoen, J., Leucker, M. (eds.) FM 2023. LNCS, vol. 14000, pp. 160–178. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-27481-7_11
38. Shi, L., Zhao, Y., Liu, Y., Sun, J., Dong, J.S., Qin, S.: A UTP semantics for communicating processes with shared variables and its formal encoding in PVS. Formal Aspects Comput. **30**(3–4), 351–380 (2018). https://doi.org/10.1007/s00165-018-0453-7
39. Smans, J., Vanoverberghe, D., Devriese, D., Jacobs, B., Piessens, F.: Shared boxes: rely-guarantee reasoning in VeriFast. Technical report, Katholieke Universiteit Leuven, Netherlands (2014). https://lirias.kuleuven.be/handle/123456789/456819
40. Song, H., Compton, K.J., Rounds, W.C.: SPHIN: a model checker for reconfigurable hybrid systems based on SPIN. In: Lazic, R., Nagarajan, R. (eds.) Proceedings of the 5th International Workshop Automated Verification of Critical Systems (AVoCS). ENTCS, vol. 145, pp. 167–183. Elsevier (2005). https://doi.org/10.1016/j.entcs.2005.10.011
41. Tarski, A.: A Decision Method for Elementary Algebra and Geometry, 2nd edn. University of California Press, Berkeley (1951). https://doi.org/10.1525/9780520348097
42. Wang, S., Zhan, N., Guelev, D.: An assume/guarantee based compositional calculus for hybrid CSP. In: Agrawal, M., Cooper, S.B., Li, A. (eds.) TAMC 2012. LNCS, vol. 7287, pp. 72–83. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29952-0_13

43. Wang, S., Zhan, N., Zou, L.: An improved HHL prover: an interactive theorem prover for hybrid systems. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 382–399. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_25
44. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects Comput. **9**(2), 149–174 (1997). https://doi.org/10.1007/BF01211617
45. Zou, L., et al.: Verifying Chinese train control system under a combined scenario by theorem proving. In: Cohen, E., Rybalchenko, A. (eds.) VSTTE 2013. LNCS, vol. 8164, pp. 262–280. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54108-7_14
46. Zwiers, J., de Bruin, A., de Roever, W.P.: A proof system for partial correctness of dynamic networks of processes. In: Clarke, E., Kozen, D. (eds.) Logic of Programs 1983. LNCS, vol. 164, pp. 513–527. Springer, Heidelberg (1984). https://doi.org/10.1007/3-540-12896-4_384
47. Zwiers, J., de Roever, W.P., van Emde Boas, P.: Compositionality and concurrent networks: soundness and completeness of a proofsystem. In: Brauer, W. (ed.) ICALP 1985. LNCS, vol. 194, pp. 509–519. Springer, Heidelberg (1985). https://doi.org/10.1007/BFb0015776

An Isabelle/HOL Formalization of the SCL(FOL) Calculus

Martin Bromberger[1], Martin Desharnais[1,2](✉), and Christoph Weidenbach[1]

[1] Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
{mbromber,desharnais,weidenbach}@mpi-inf.mpg.de
[2] Graduate School of Computer Science, Saarland Informatics Campus, Saarbrücken, Germany

Abstract. We present an Isabelle/HOL formalization of Simple Clause Learning for first-order logic without equality: SCL(FOL). The main results are formal proofs of soundness, non-redundancy of learned clauses, termination, and refutational completeness. Compared to the unformalized version, the formalized calculus is simpler and more general, some results such as non-redundancy are stronger and some results such as non-subsumption are new. We found one bug in a previously published version of the SCL Backtrack rule. Compared to related formalizations, we introduce a new technique for showing termination based on non-redundant clause learning.

Keywords: interactive theorem proving · automated theorem proving · first-order logic · CDCL · SCL · non-redundant clause learning

1 Introduction

The SCL ("Clause Learning from Simple Models" or simply "Simple Clause Learning") family of calculi lifts a conflict-driven clause learning (CDCL) approach to first-order logic: SCL(FOL) is for first-order logic without equality [8,10], SCL(T) is for first-order logic with theories [6], SCL(EQ) is for first-order logic with equality [12], and HSCL is for exhaustive partial models exploration in first-order logic without equality [7]. In its original formulation [10], SCL(FOL) required exhaustive propagation and a precise strategy for the application of the rules in order to learn non-redundant clauses. This was improved upon by SCL(T) [6] by dropping exhaustive propagation and weakening the strategy, i.e., any run according to the strategy in [10] is also a run according to the strategy in [6]. The SCL(FOL) version presented in Bromberger et al. [8] integrates those changes and additionally refines the Backtrack rule.

We present an Isabelle/HOL formalization of the non-executable specification of SCL(FOL) based on and developed in parallel to Bromberger et al. The

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 116–133, 2023.
https://doi.org/10.1007/978-3-031-38499-8_7

main results are soundness, non-redundancy of learned clauses, termination, and refutational completeness. In contrast to the goal of Bromberger et al. to guide toward an implementation, our goal is to be as simple and general as possible. For that, we (i) simplified the calculus (e.g., no more explicity tracking of decision levels), (ii) generalized the calculus (e.g., multiple acceptable positions in the Backtrack rule), (iii) strengthened existing theorems (e.g., Theorem 11 on non-redundancy), and (iv) proved new theorems (e.g., Corollary 12 on non-subsumption).

This work is part of the IsaFoL (Isabelle Formalization of Logic) effort [2], which aims at developing a library of results about logical calculi. The Isabelle theory files are available in the *Archive of Formal Proofs* (AFP) [9] and amount to more than 11 000 lines of source text. They build heavily upon many other entries of the AFP: (i) First_Order_Terms [17] for first-order terms, term substitutions, and MGU; (ii) Ordered_Resolution_Prover [14–16] for the clausal calculus, clause substitutions, Herbrand interpretation, and compactness of first-order logic; and (iii) Saturation_Framework_Extensions [5,18] for entailment of the clausal calculus. We contributed many lemmas and definitions back to both the Isabelle distribution and the aforementioned AFP entries (e.g., over 50 to First_Order_Terms). We made heavy use of the Isar language [19] to write structured proofs, the Sledgehammer tool [13] for proof automation, and locales [1]—Isabelle's parameterized module system—to structure our development and reuse existing components from the AFP entries. To ease associating the main results in this paper with their counterparts in the Isabelle development, names in `monospace` are taken verbatim from the formalization.

The formalization follows the basic ideas of the existing formalizations of the first-order resolution calculus [16] and propositional CDCL calculi [3,4]. Compared to propositional logic, first-order logic adds a number of challenges: the extra term level requires to consider variables, substitutions, groundings, and the concept of factorization. To preserve completeness, propagation of ground literals must not be exhaustive anymore, resulting in a level-wise exploration w.r.t. a bounding atom. Inside this bound, the calculus always terminates. If one level does not suffice to find a refutation, the bound can be increased and exploration can be continued. For unsatisfiable formulas, we prove the existence of a bound sufficient to derive $\bot$, which guarantees that only finitely many levels need to be explored.

The paper is now organized as follows. Section 2 recaps the SCL(FOL) calculus from Bromberger et al. as the basis of our formalization presented in Sect. 3. We first present the Isabelle formalization of the abstract rules of the SCL(FOL) calculus. Then we prove invariants preserved by the rules starting from the initial state, Lemma 1. Subsequently, we prove soundness, Theorem 7, non-redundancy of learned clauses, Theorem 11, termination with respect to a fixed bound, Theorem 18, and finally refutational completeness with respect to an appropriate bound, Theorem 20. We discuss important aspects of the formalization and proof ideas here and refer the reader to the formalization for more details. The paper ends with a short conclusion of the obtained results.

2 The SCL(FOL) Calculus

We shortly repeat basic first-order logic notions and the SCL(FOL) calculus presented in Bromberger et al. We consider an untyped, first-order logic without equality. A *term* is defined inductively as either a variable x or a function application $f(\overrightarrow{t})$ for a constant f and a (possibly-empty) list of terms $\overrightarrow{t}$. An *atom* is a predicate symbol applied to a list of term arguments. A *literal* is either a positive atom A or a negative atom $\neg A$. For literals we write L or K. The atom of a literal may be selected with $\operatorname{atom}(A) = A$ and $\operatorname{atom}(\neg A) = A$. The complement of a literal is defined as $\operatorname{comp}(A) = \neg A$ and $\operatorname{comp}(\neg A) = A$. A disjunctive *clause* is a finite multiset of literals. For clauses we write C or D. We use the syntax $L \vee C$ and $C \vee D$ synonymously with the multiset sums $\{L\} + C$ and $C + D$ respectively. We also use the syntax $\bot$ synonymously with the empty multiset $\{\}$. All variables in clauses are to be understood as universally quantified.

Substitutions are total unary functions from variables to terms. A substitution σ may be applied to a variable x, a term t, an atom A, a literal L, or a clause C, denoted $x\sigma$, $t\sigma$, $A\sigma$, $L\sigma$, or $C\sigma$ respectively. Substitution application is left-associative, i.e., $C\sigma_1\sigma_2 = (C\sigma_1)\sigma_2$. The domain of a substitution σ is defined as $\operatorname{dom}(\sigma) = \{x \mid x\sigma \neq x\}$. The composition of two substitutions σ_1 and σ_2 is defined as the function $\sigma_1 \circ \sigma_2 = (\lambda x.\, x\sigma_1\sigma_2)$. A substitution γ is a *grounding* for a term t, an atom A, a literal L, or a clause C if $t\gamma$, $A\gamma$, $L\gamma$, or $C\gamma$ are respectively ground, i.e., if they do not contain variables. A substitution ρ is a *renaming* if it is injective and $x\rho$ is a variable for all variables x. The *inverse* of a renaming ρ is any function ρ^{-1} from terms to variables such that $\rho^{-1}\,(x\rho) = x$ for all variables x. The *restriction* of a substitution σ to a set of variables V is defined as the function $(\lambda x.\ \text{if } x \in V \text{ then } x\sigma \text{ else } x)$. A substitution σ is *idempotent* if $\sigma \circ \sigma = \sigma$. A substitution υ is a *unifier* for a set of terms T if $t_1\upsilon = t_2\upsilon$ for all terms $t_1 \in T$ and $t_2 \in T$. A substitution μ is a *most general unifier* (MGU) for a set of terms T if μ is a unifier for T and there exists a substitution σ such that $\mu \circ \sigma = \upsilon$ for all unifiers υ for T. A substitution μ is an *idempotent, most general unifier* (IMGU) for a set of terms T if μ is a unifier for T and $\mu \circ \upsilon = \upsilon$ for all unifiers υ for T; note that μ is an IMGU iff it is both idempotent and a MGU.

When formalizing logical calculi, IMGUs are preferable because they allow to apply groundings to a term both directly and after applying an IMGU, i.e., $t\gamma = t\mu\gamma$ for all terms t, groundings γ, and IMGU μ. Non-idempotent MGU do not have this property as the following counter-example shows. Consider the terms $t_1 = f(x, y, z)$ and $t_2 = f(w, y, z)$, the grounding $\gamma = \{x \mapsto a,\ y \mapsto b,\ z \mapsto c,\ w \mapsto a\}$, and the non-idempotent MGU $\mu = \{x \mapsto w,\ y \mapsto z,\ z \mapsto y\}$ where x, y, z, w are variables and a, b, c are ground constants, then we have $t_1\gamma = f(a, b, c) \neq f(a, c, b) = t_1\mu\gamma$. In published literature, an IMGU is often meant instead of an MGU; the idempotency requirement is often kept implicit because standard implementations for computing MGUs actually produce IMGUs.

The function $\operatorname{gnd}(C) = \{C\gamma \mid C\gamma \text{ is ground}\}$ expresses the set of all groundings of a clause C. The function $\operatorname{gnd}(N) = (\bigcup C \in N.\ \operatorname{gnd}(C))$ expresses the set of all groundings of a set of clauses N; its subset whose clauses are restricted to atoms less than or equal to a bound β w.r.t. an order $\prec_B$ is defined as $\text{gnd}^{\preceq_B \beta}(N) = \{C \in \text{gnd}(N) \mid \forall L \in C.\ \text{atom}(L) \preceq_B \beta\}$. Note that $\text{gnd}(\text{gnd}(N)) = \text{gnd}(N)$. The strict order $\prec_B$ is total on ground literals and is such that for each β there are only finitely many literals L with $L \prec_B \beta$. An example of such an order could be KBO without zero-weight symbols. Note that LPO does not satisfy the last condition of a $\prec_B$ order although it is a well-founded and total order.

Herbrand entailment is defined as $(I \models_{\mathcal{H}} N \longleftrightarrow (\forall C \in N.\ I \models_{\mathcal{H}} C))$ for a set of clauses N, $(I \models_{\mathcal{H}} C \longleftrightarrow (\exists L \in C.\ I \models_{\mathcal{H}} L))$ for a clause C, $(I \models_{\mathcal{H}} A \longleftrightarrow A \in I)$, and $(I \models_{\mathcal{H}} \neg A \longleftrightarrow A \notin I)$ for a literal with atom A; note that the symbol $\models_{\mathcal{H}}$ is overloaded. Ground entailment is defined as $(N_1 \models_{\mathcal{G}} N_2 \longleftrightarrow (\forall I.\, I \models_{\mathcal{H}} N_1 \longrightarrow I \models_{\mathcal{H}} N_2))$. First-order entailment is defined as $(N_1 \models N_2 \longleftrightarrow \text{gnd}(N_1) \models_{\mathcal{G}} \text{gnd}(N_2))$. A set of ground clauses N is satisfiable if there exists a Herbrand interpretation I such that $I \models_{\mathcal{H}} N$; otherwise, it is unsatisfiable.

An annotated literal is the pairing of a literal with an annotation. We call it a *decision literal* when the annotation is a natural number n indicating the literal's level (i.e., that it is the nth decision) and a *propagation literal* when the annotation is a closure of the clause the literal originated from. The literal of an annotated literal $\mathcal{K}$ is denoted $\text{lit}(\mathcal{K})$ and the annotation is denotated $\text{ann}(\mathcal{K})$. The level of a clause is the maximum level of its literals. A *trail* is a finite sequence of annotated ground literals: it grows from left to right. The empty trail is written ϵ and appending a new annotated literal $\mathcal{K}$ to a trail Γ is written $\Gamma,\mathcal{K}$. The concatenation of two trails Γ_1 and Γ_2 is written Γ_2, Γ_1. A trail Γ can be converted to a set with $\text{set}(\Gamma)$.

A literal L is true under trail Γ if $L \in \{\text{lit}(\mathcal{K}) \mid \mathcal{K} \in \text{set}(\Gamma)\}$. A literal L is false under trail Γ if $\text{comp}(L) \in \{\text{lit}(\mathcal{K}) \mid \mathcal{K} \in \text{set}(\Gamma)\}$. A literal L is defined in a trail Γ if L is true or false under Γ; otherwise, it is undefined. A clause C is true under trail Γ if $(\exists L \in C.\ L$ is true under $\Gamma)$. A clause C is false under trail Γ if $(\forall L \in C.\ L$ is false under $\Gamma)$. A clause C is defined in a trail Γ if $(\forall L \in C.\ L$ is defined in $\Gamma)$; otherwise, it is undefined.

The SCL(FOL) calculus is defined as a transition system operating on states $(\Gamma; N; U; \beta; k; \mathcal{C})$ where Γ is a trail, N is a finite set of initial clauses, U is a finite set of learned clauses, β is a bounding atom restricting the considered ground literals, k is a natural number counting the number of decisions taken in Γ, and $\mathcal{C}$ is either $\top$ or a clause closure $(C; \gamma)$ such that $C\gamma$ is ground and false in Γ. The initial state is $(\epsilon; N; \emptyset; \beta; 0; \top)$ for some initial clause set N and bound β.

The transition relation $\Rightarrow_{\text{SCL}}$ is a mapping between states. The rules below are from Bromberger et al. and serve as a reference for the Isabelle formalization described in Sect. 3.

Propagate $(\Gamma; N; U; \beta; k; \top) \ \Rightarrow_{\text{SCL}} \ (\Gamma, L\gamma^{((C_0 \vee L)\mu;\gamma)}; N; U; \beta; k; \top)$
if $(C \vee L) \in (N \cup U)$, $C = C_0 \vee C_1$, $C_1\gamma = L\gamma \vee \cdots \vee L\gamma$, $C_0\gamma$ does not contain $L\gamma$, μ is the IMGU of the literals in C_1 and L, $(C \vee L)\gamma$ is ground, $(C \vee L)\gamma \prec_B \{\beta\}$, $C_0\gamma$ false under Γ, and $L\gamma$ is undefined in Γ.

Decide $(\Gamma; N; U; \beta; k; \top) \ \Rightarrow_{\text{SCL}} \ (\Gamma, L\gamma^{k+1}; N; U; \beta; k+1; \top)$
if $L \in C$ for a $C \in (N \cup U)$, $L\gamma$ is a ground literal undefined in Γ, and $L\gamma \prec_B \beta$.

Conflict $(\Gamma; N; U; \beta; k; \top) \Rightarrow_{\text{SCL}} (\Gamma; N; U; \beta; k; (C; \gamma))$
if $C \in (N \cup U)$, $C\gamma$ is false under Γ for a grounding substitution γ.

These rules construct a (partial) model via the trail Γ for $N \cup U$ until a conflict, i.e., a clause false under Γ is found. The above rules always terminate, because there are only finitely many ground literals L with $L \prec_B \beta$. It might be necessary to successively increase β for full refutational completeness.

Skip $(\Gamma, K; N; U; \beta; k; (C; \gamma)) \Rightarrow_{\text{SCL}} (\Gamma; N; U; \beta; k - i; (C; \gamma))$
if comp(K) does not occur in $C\gamma$, if K is a decision literal then $i = 1$; otherwise, $i = 0$.

Factorize $(\Gamma; N; U; \beta; k; (C \vee L \vee L'; \gamma)) \Rightarrow_{\text{SCL}} (\Gamma; N; U; \beta; k; ((C \vee L)\mu; \gamma))$
if $L\gamma = L'\gamma$ and $\mu = \text{IMGU}(L, L')$.

Note that this rule may be used multiple times if the conflicting clause contains more than two duplicates of a given literal or if multiple distinct literals have duplicates.

Resolve $(\Gamma, K\gamma_D{}^{(D \vee K; \gamma_D)}; N; U; \beta; k; (C \vee L; \gamma_C))$
$\Rightarrow_{\text{SCL}} (\Gamma, K\gamma_D{}^{(D \vee K; \gamma_D)}; N; U; \beta; k; ((C \vee D)\mu; \gamma_C \circ \gamma_D))$
if $K\gamma_D = \text{comp}(L\gamma_C)$, $\mu = \text{IMGU}(K, \text{comp}(L))$.

The clauses $D \vee K$ and $C \vee L$ are assumed to have disjoint variables.

Backtrack $(\Gamma_0, K, \Gamma_1, \text{comp}(L\gamma)^k; N; U; \beta; k; (C \vee L; \gamma))$
$\Rightarrow_{\text{SCL}} (\Gamma_0; N; U \cup \{C \vee L\}; \beta; j; \top)$
if $C\gamma$ is of level $i' < k$, and Γ_0, K is the minimal trail subsequence such that there is a grounding substitution γ' with $(C \vee L)\gamma'$ is false under Γ_0, K but not in Γ_0, and Γ_0 is of level j.

The clause $C \vee L$ added by the rule Backtrack to U is called a *learned clause*. The empty clause $\bot$ can only be generated by rule Resolve or be already present in N, hence, as usual for CDCL-style calculi, the generation of $\bot$ together with the clauses in $N \cup U$ represent a resolution refutation.

A sequence of SCL rule applications is called a *reasonable run* if the rule Decide does not enable an immediate application of rule Conflict. A sequence of SCL rule applications is called a *regular run* if it is a reasonable run and the rule Conflict has precedence over all other rules.

3 Formalization of the SCL(FOL) Calculus

The formalization introduces some new concepts absent from Sect. 2. A multiset C can be converted to a set, i.e., without duplicates, with set(C). The multiplicity of an element x in a multiset C is denoted by count(C, x). The cardinality of a multiset—the sum of the multiplicities of its elements—is denoted by $|C|$. The multiset whose only element is x with multiplicity n is denoted by repeat(n, x); note that count(repeat$(n, x), x) = n$, and set(repeat$(n, x)) = \{x\}$ if $n > 0$. The multiset extension of an order on literals extends the order to multisets containing literals; we use the Huet-Oppen specification [11], one of several equivalent alternatives for this extension. The *adaptation* of a substitution σ to a renaming ρ is a function whose domain is the renamed domain of σ and whose codomain is the same as σ; it is defined as the function $(\lambda x.\, \text{if } x \in \{y\rho \mid y \in \text{dom}(\sigma)\} \text{ then } (\rho^{-1}\, x)\sigma \text{ else } x)$. A substitution γ is a *merged grounding* of a grounding γ_A for a set of variables A and a grounding γ_B for a set of variables B if $(A \cap B = \{\} \longrightarrow (\forall x \in A.\ x\gamma_A \text{ is ground}) \longrightarrow (\forall x \in B.\ x\gamma_B \text{ is ground}) \longrightarrow (\forall x \in A.\ x\gamma = x\gamma_A) \wedge (\forall x \in B.\ x\gamma = x\gamma_B))$; an example of a function that fulfills this specification is $(\lambda x.\, \text{if } x \in A \text{ then } x\gamma_A \text{ else } x\gamma_B)$. The length of a trail Γ is denoted by $|\Gamma|$. The nth right-most element of a trail Γ is denoted by $\Gamma[n]$; we use zero-based indexing where the right-most element is the 0th element. The Herbrand interpretation of a trail Γ is defined as $\mathcal{HI}(\Gamma) = (\bigcup \mathcal{K} \in \text{set}(\Gamma).\ \text{case lit}(\mathcal{K}) \text{ of } A \Rightarrow \{A\} \mid \neg A \Rightarrow \{\})$.

The formalization also changes some existing concepts. No distinction is made between *atoms* and terms, so first-order terms are used everywhere in place of atoms. The level annotation of a *decision literal* is not required anymore and replaced by a † marker, it is now written $K = (K; \dagger)$ for some literal K. A *propagation literal* is written $(K\gamma_D)^{(K;D;\gamma_D)} = (K\gamma_D; (D; K; \gamma_D))$ for some literal K, clause D, and grounding γ_D. Note that the propagated literal is explicitly separated from its clause in the closure annotation; this eases the formulation of the additional invariants 5 and 6 of Lemma 1., that the respective clause is always false under the respective trail. For the *trail* Γ,$\mathcal{K}$, the Isabelle formalization uses the constructor `List.Cons` $\mathcal{K}\,\Gamma$ which actually grows from right to left. However, we keep the well-established left-to-right convention in this paper because it significantly eases the presentation. An state is a tuple $(\Gamma; U; \mathcal{C})$ where Γ is a trail, U is a finite set of learned clauses, and $\mathcal{C}$ is an optional clause closure. The individual components can be selected with $\text{trail}((\Gamma; U; \mathcal{C})) = \Gamma$, $\text{learned}((\Gamma; U; \mathcal{C})) = U$, and $\text{conflict}((\Gamma; U; \mathcal{C})) = \mathcal{C}$. The *initial state* is $(\epsilon; \{\}; \top)$, i.e., empty trail, no learned clauses, and no conflicting closure. The finite set of initial clauses N and the bounding atom β are no longer stored in the state but are rather parameters of the transition relation; this was done to highlight the fact that they are never modified by any rule. The natural number k counting the number of decisions, used in Sect. 2 to determine an appropriate backtracking point, turned out not to be necessary and was dropped entirely. We assume the existence of a binary relation on atoms $\prec_B$ such that $(\forall \beta.\ \{t \mid t \prec_B \beta\} \text{ is finite})$ but dropped the requirement for $\prec_B$ to be a strict order total on ground terms. We also don't lift $\prec_B$ to literals and clauses, but always use it at the atom level. We define the relation $\preceq_B$ as the reflexive closure of $\prec_B$.

The transition relation $\Rightarrow_{\text{SCL}}^{N,\beta}$ is a binary predicate between states and is parameterized by the finite set N of initial clauses and the bounding atom β. It is defined as the disjunction of the following rules. Following each rule, we highlight the main differences from Sect. 2 not already covered.

Propagate $(\Gamma; U; \top) \Rightarrow_{\text{Propagate}}^{N,\beta} (\Gamma, (L\mu\gamma)^{(L\mu; C_0\mu; \gamma)}; U; \top)$
if $(L \vee C) \in (N \cup U)$, γ is a grounding for $L \vee C$, $(\forall K \in (L \vee C).\ \text{atom}(K\gamma) \preceq_B \beta)$, $C_0 = \{K \in C \mid K\gamma \neq L\gamma\}$, $C_1 = \{K \in C \mid K\gamma = L\gamma\}$, $C_0\gamma$ is false under Γ, $L\gamma$ is undefined in Γ, and μ is an IMGU for all terms in $\{\text{atom}(K) \mid K \in (L \vee C_1)\}$.

Compared to Sect. 2, we express the splitting of C into C_0 and C_1 formally as set operations and replace $\prec_B$ with $\preceq_B$. This replacement has no effect on the results but allowing the bound β to be in $\mathrm{gnd}^{\preceq_B \beta}(N)$ eases the proof of Lemma 21, where the largest element of the (finite) unsatisfiable core is directly used as new bound. There are also situations where the maximal element of a signature is required to derive a contradiction: a non-strict bound requires to artificially extend the signature while a non-strict bound does not.

Decide $(\Gamma; U; \top) \Rightarrow^{N,\beta}_{\text{Decide}} (\Gamma, (L\gamma); U; \top)$
if $(L \vee C) \in N$, γ is a grounding for L, $L\gamma$ is undefined in Γ, and $\mathrm{atom}(L\gamma) \preceq_B \beta$.

Compared to Sect. 2, we replace $\prec_B$ with $\preceq_B$ and take the decision literal from N instead of $N \cup U$. The ground instances of literals of U are a subset of the ground instances of literals of N so it is redundant to also consider U here.

Conflict $(\Gamma; U; \top) \Rightarrow^{N,\beta}_{\text{Conflict}} (\Gamma; U; (C; \gamma))$
if $C \in (N \cup U)$, γ is a grounding for C, and $C\gamma$ is false under Γ.

Skip $(\Gamma, \mathcal{K}; U; (C; \gamma)) \Rightarrow^{N,\beta}_{\text{Skip}} (\Gamma; U; (C; \gamma))$
if $\mathrm{comp}(\mathrm{lit}(\mathcal{K})) \notin C\gamma$.

Factorize $(\Gamma; U; (L' \vee L \vee C; \gamma)) \Rightarrow^{N,\beta}_{\text{Factorize}} (\Gamma; U; ((L \vee C)\mu; \gamma))$
if $L\gamma = L'\gamma$ and μ is the IMGU for the terms $\mathrm{atom}(L)$ and $\mathrm{atom}(L')$.

Resolve $(\Gamma; U; (L \vee C; \gamma_C)) \Rightarrow^{N,\beta}_{\text{Resolve}} (\Gamma; U; ((C\rho_C \vee D\rho_D)\mu; \gamma))$
if $\Gamma = \Gamma', (K\gamma_D)^{(K;D;\gamma_D)}$, and $K\gamma_D = \mathrm{comp}(L\gamma_C)$, ρ_C and ρ_D are renamings such that the variables of $(L \vee C)\rho_C$ and $(K \vee D)\rho_D$ are disjoint, μ is the IMGU for the terms $\mathrm{atom}(L)\rho_C$ and $\mathrm{atom}(K)\rho_D$, γ'_C and γ'_D are adaptations of γ_C and γ_D to the renamings ρ_C and ρ_D respectively, and γ is a merged grounding of γ'_C for the variables of $(L \vee C)\rho_C$ and γ'_D for the variables of $(K \vee D)\rho_D$.

Note that the definition of merged grounding implies the following equalities: $\mu \circ \gamma = \gamma$, $L\rho_C\gamma = L\gamma_C$, $C\rho_C\gamma = C\gamma_C$, $K\rho_D\gamma = K\gamma_D$, and $D\rho_D\gamma = D\gamma_D$.

Compared to Sect. 2, we explicitly rename the merged clauses to avoid variable-name clashes instead of assuming disjoint variables, and use an abstract specification for the merged grounding instead of forcing substitution composition. The latter makes our rule more general by allowing more freedom to an implementation.

Backtrack $(\Gamma, \Gamma', K; U; (L \vee C; \gamma)) \Rightarrow^{N,\beta}_{\text{Backtrack}} (\Gamma; \{L \vee C\} \cup U; \top)$
if $K = \mathrm{comp}(L\gamma)$ and $(\nexists \gamma'.\ (L \vee C)\gamma'$ is ground and false under $\Gamma)$.

Compared to Sect. 2, we allow backtracking to any non-conflicting trail instead of specifying the position. This makes our rule more general by, again, allowing more freedom to an implementation. The minimally backtracking strategy introduced in Definition 4 brings back equivalence to the Backtrack rule of Sect. 2.

Isabelle Technicalities. We define the SCL rules in the `scl_fol_calculus` locale. It fixes an abstract binary relation $\prec_B$ as a locale parameter and assumes that it bounds a finite number of atoms. It also fixes an abstract function to generate variable renamings as a locale parameter and assumes its correctness; this function is not required for the specification of the calculus but is required in multiple proofs. Most of the following definitions and theorems are in the context of this locale. Each SCL rule is defined separately as an inductive predicate. Having separate definitions allows to refer to the rules individually in subsequent definitions and theorems. Using inductive predicates, as opposed to plain definitions, is convenient because Isabelle automatically generates some useful introduction and elimination lemmas, and configures structured Isar syntax for case analysis.

From the SCL rules, we can prove a number of invariants about states. Most of them are intuitive while few are technicalities of the Isabelle formalization. We will use the invariants as hypotheses for many of the main lemmas and theorems.

Lemma 1 (`scl_state_invariants`). *Let* $(\Gamma; U; \mathcal{C})$ *be an state w.r.t.* $\Rightarrow_{SCL}^{N,\beta}$. *The following invariants hold for the initial state* $(\epsilon; \{\}; \top)$ *and are each individually preserved by the SCL rules.*

1. *All annotated literals in* Γ *are ground.*
   - $\forall K \in \{\mathit{lit}(\mathcal{K}) \mid \mathcal{K} \in \mathit{set}(\Gamma)\}.\ K$ *is a ground literal*
2. *The atoms of all annotated literals in* Γ *are* $\preceq_B \beta$.
   - $\forall K \in \{\mathit{lit}(\mathcal{K}) \mid \mathcal{K} \in \mathit{set}(\Gamma)\}.\ \mathit{atom}(K) \preceq_B \beta$
3. *All annotated literals in* Γ *are undefined in their respective subtrail of* Γ.
   - $\forall \Gamma'\, \mathcal{K}\, \Gamma''.\ \Gamma = \Gamma', \mathcal{K}, \Gamma'' \longrightarrow \mathit{lit}(\mathcal{K})$ *is undefined in* Γ'
4. *All closures in* Γ *and* $\mathcal{C}$ *are ground.*
   - $\forall \mathcal{K} \in \mathit{set}(\Gamma).\ \forall D\, K\, \gamma.\ \mathcal{K} = (K\gamma)^{(K;D;\gamma)} \longrightarrow D\gamma$ *is ground*
   - $\forall C\, \gamma.\ \mathcal{C} = (C;\gamma) \longrightarrow C\gamma$ *is ground*
5. *All closures in* Γ *and* $\mathcal{C}$ *are false under their respective subtrail of* Γ.
   - *invariant 4. holds*
   - $\forall D\, K\, \gamma\, \Gamma'\, \Gamma''.\ \Gamma = \Gamma', (K\gamma)^{(K;D;\gamma)}, \Gamma'' \longrightarrow D\gamma$ *is false under* Γ'
   - $\forall C\, \gamma.\ \mathcal{C} = (C;\gamma) \longrightarrow C\gamma$ *is false under* Γ
6. *All propagated literals in* Γ *are the grounding of the non-ground literal in their closure annotations.*
   - $\forall \mathcal{K} \in \mathit{set}(\Gamma).\ \forall D\, K\, \gamma.\ \mathit{ann}(\mathcal{K}) = (D; K; \gamma) \longrightarrow \mathit{lit}(\mathcal{K}) = K\gamma$
7. *The complements of all propagated literals in* Γ *are absent from their closure annotation.*
   - $\forall \mathcal{K} \in \mathit{set}(\Gamma).\ \forall D\, K\, \gamma.\ \mathcal{K} = (K\gamma)^{(K;D;\gamma)} \longrightarrow \mathit{comp}(K\gamma) \notin D\gamma$
8. *All literals of the clauses in* Γ*'s propagating clauses,* U*, and* $\mathcal{C}$ *have a corresponding, more general literal in* N.
   - $\forall D \in \{D \mid (K\gamma)^{(K;D;\gamma)} \in \mathit{set}(\Gamma)\} \cup U \cup (\mathit{if}\ \mathcal{C} = (C;\gamma)\ \mathit{then}\ \{C\}\ \mathit{else}\ \{\}).$
     $\forall K \in D.\ \exists D' \in N.\ \exists K' \in D'.\ \exists \sigma.\ K'\sigma = K$
9. *All annotated literals in* Γ *have a corresponding more general literal either in* N *or in* U.
   - $\forall \mathcal{K} \in \mathit{set}(\Gamma).\ \exists L \in N \cup U.\ \exists \sigma.\ L\sigma = \mathit{lit}(\mathcal{K})$
10. *All clauses in* Γ*,* U*, and* $\mathcal{C}$ *are entailed by* N.
    - $\forall \mathcal{K} \in \mathit{set}(\Gamma).\ \forall D\, K\, \gamma.\ \mathcal{K} = (K\gamma)^{(K;D;\gamma)} \longrightarrow N \models \{K \vee D\}$
    - $N \models U$
    - $\forall C\, \gamma.\ \mathcal{C} = (C;\gamma) \longrightarrow N \models \{C\}$

The SCL calculus is defined as a transition system where many decisions are deferred to strategies. A *strategy* specifies a transition system whose transitions are a subset of those from an existing transition system. We say that a strategy $\mathcal{S}$ *restricts* a transition system $\mathcal{T}$ (or symmetrically that $\mathcal{T}$ is *restricted* by $\mathcal{S}$) if $(\forall x\, y.\ \mathcal{S}\, x\, y \longrightarrow \mathcal{T}\, x\, y)$. Note that strategies can be chained to iteratively apply more restrictions.

We define the reasonable and regular strategies restricting the $\Rightarrow^{N,\beta}_{\mathrm{SCL}}$ relation in order to prove the main results of this paper.

Definition 2. *The reasonable strategy* $\Rightarrow^{N,\beta}_{\mathit{Rea\text{-}SCL}}$ *restricts the SCL calculus by preventing decisions that immediately lead to a conflict. Such situations could be replaced by a propagation. Formally:*

$$S \Rightarrow^{N,\beta}_{\mathit{Rea\text{-}SCL}} S' \longleftrightarrow S \Rightarrow^{N,\beta}_{\mathit{SCL}} S' \wedge (S \Rightarrow^{N,\beta}_{\mathit{Decide}} S' \longrightarrow (\nexists S''.S' \Rightarrow^{N,\beta}_{\mathit{Conflict}} S''))$$

Definition 3. *The regular strategy* $\Rightarrow^{N,\beta}_{\mathit{Reg\text{-}SCL}}$ *restricts the reasonable strategy by prioritizing the conflict rule to any other. Formally:*

$$S \Rightarrow^{N,\beta}_{\mathit{Reg\text{-}SCL}} S' \longleftrightarrow S \Rightarrow^{N,\beta}_{\mathit{Rea\text{-}SCL}} S' \wedge ((\exists S''.\ S \Rightarrow^{N,\beta}_{\mathit{Conflict}} S'') \longrightarrow S \Rightarrow^{N,\beta}_{\mathit{Conflict}} S')$$

While not required for the coming results, we also define the minimally backtracking strategy to express the constraint on the backtracking position found in Sect. 2.

Definition 4. *The minimally backtracking strategy* $\Rightarrow^{N,\beta}_{\mathit{Min\text{-}Bac\text{-}SCL}}$ *restricts the regular strategy by requiring that backtracking removes the shortest possible suffix of the trail. Formally:*

$$\begin{aligned} S \Rightarrow^{N,\beta}_{\mathit{Min\text{-}Bac\text{-}SCL}} S' \longleftrightarrow\ & S \Rightarrow^{N,\beta}_{\mathit{Reg\text{-}SCL}} S' \wedge (S \Rightarrow^{N,\beta}_{\mathit{Backtrack}} S' \longrightarrow \\ & \mathit{trail}(S') \textit{ is the longest prefix of } \mathit{trail}(S) \\ & \textit{not in conflict with the learned clause}) \end{aligned}$$

All three strategies build on one-another and ultimately restrict the SCL relation. We can express this formally as implications, of which the first can be used to show that coming results (e.g., Corollaries 13 and 19) also hold for the minimally backtracking strategy.

Lemma 5 (`strategy_restrictions`). *The minimally backtracking strategy restricts the regular strategy, which restricts the reasonable strategy, which restricts the SCL calculus. Formally:*

- $\forall N\, \beta\, S\, S'.\ S \Rightarrow^{N,\beta}_{\mathit{Min\text{-}Bac\text{-}SCL}} S' \longrightarrow S \Rightarrow^{N,\beta}_{\mathit{Reg\text{-}SCL}} S'$
- $\forall N\, \beta\, S\, S'.\ S \Rightarrow^{N,\beta}_{\mathit{Reg\text{-}SCL}} S' \longrightarrow S \Rightarrow^{N,\beta}_{\mathit{Rea\text{-}SCL}} S'$
- $\forall N\, \beta\, S\, S'.\ S \Rightarrow^{N,\beta}_{\mathit{Rea\text{-}SCL}} S' \longrightarrow S \Rightarrow^{N,\beta}_{\mathit{SCL}} S'$

The bounding atom β restricts the calculus to only consider the finitely many ground atoms less than or equal to β w.r.t. $\prec_B$; this will play an important role in the termination proof. When SCL terminates, it either derived a contradiction, or it found a model for the bounded groundings of the initial clauses. Because β is usually chosen heuristically, the model might be unsatisfactory for the considered use case and one may want to continue execution with a bigger bound. This is allowed if the new bound properly extends the previous bound β w.r.t. $\preceq_B$.

Theorem 6 (`monotonicity_wrt_bound`). *If the ground atoms bound by β are a subset of the ground atoms bound by β', formally if $(\forall A.\ A$ is ground $\longrightarrow A \preceq_B \beta \longrightarrow A \preceq_B \beta')$, then the SCL, reasonable SCL, regular SCL, and minimally backtracking transitions w.r.t. β are also transitions w.r.t. β', formally*

- $\forall N\, S\, S'.\ S \Rightarrow^{N,\beta}_{SCL} S' \longrightarrow S \Rightarrow^{N,\beta'}_{SCL} S'$,
- $\forall N\, S\, S'.\ S \Rightarrow^{N,\beta}_{Rea\text{-}SCL} S' \longrightarrow S \Rightarrow^{N,\beta'}_{Rea\text{-}SCL} S'$,
- $\forall N\, S\, S'.\ S \Rightarrow^{N,\beta}_{Reg\text{-}SCL} S' \longrightarrow S \Rightarrow^{N,\beta'}_{Reg\text{-}SCL} S'$, *and*
- $\forall N\, S\, S'.\ S \Rightarrow^{N,\beta}_{Min\text{-}Bac\text{-}SCL} S' \longrightarrow S \Rightarrow^{N,\beta'}_{Min\text{-}Bac\text{-}SCL} S'$.

Theorem 6 implies that all properties w.r.t. a bound β also hold w.r.t. a compatible bound β'. Its hypothesis is fulfilled if $\preceq_B$ is transitive on ground atoms, β and β' are ground atoms, and $\beta \preceq_B \beta'$. The bounding atom could even be increased at any point in an SCL run, not just when the calculus terminated.

The different rules and strategies considered so far express a single step of computation for the SCL calculus; they offer a good level of granularity to both understand and mechanize the details of the calculus. But many results of the following sections ought to express properties of the calculus as a whole. We express such results in terms of a run from the initial state. A *run* is the reflexive, transitive closure of a rule or strategy, e.g. $S\,(\Rightarrow^{N,\beta}_{\mathrm{SCL}})^*\,S'$ is an SCL run from the state S to the state S'.

The soundness of the individual SCL rules is shown by invariant 10. We now consider the soundness of terminating runs of the SCL calculus as a whole.

Theorem 7 (`correct_termination`). *Let $S = (\Gamma; U; \mathcal{C})$ be a state w.r.t. $\Rightarrow^{N,\beta}_{SCL}$. If invariants 2, 3, 5, 6 and 10 hold for S, and if S is a stuck state with some restrictions, formally if*

- $\nexists S'.\ S \Rightarrow^{N,\beta}_{Propagate} S'$,
- $\nexists S'.\ S \Rightarrow^{N,\beta}_{Decide} S' \wedge (\nexists S''.\ S' \Rightarrow^{N,\beta}_{Conflict} S'')$,
- $\nexists S'.\ S \Rightarrow^{N,\beta}_{Conflict} S'$,
- $\nexists S'.\ S \Rightarrow^{N,\beta}_{Skip} S'$,
- $\nexists S'.\ S \Rightarrow^{N,\beta}_{Resolve} S'$,
- $\nexists S'.\ S \Rightarrow^{N,\beta}_{Backtrack} S'$ *and the backtracking is minimal,*

then either the conflicting clause $\bot$ has been derived and the groundings $gnd(N)$ of the initial clauses N are unsatisfiable, or there is no conflicting clause and the groundings $gnd^{\preceq_B \beta}(N)$ of the initial clauses N are satisfiable by the trail, formally either

- $(\exists \gamma.\ \mathcal{C} = (\bot; \gamma)) \wedge (\nexists I.\ I \models_{\mathcal{H}} gnd(N))$, *or*

- $\mathcal{C} = \top \wedge \mathcal{HI}(\Gamma) \models_{\mathcal{H}} gnd^{\preceq_B \beta}(N)$.

Note that no hypothesis restricts the usage of the Factorize rule because it is an optional step of conflict resolution that has no impact on satisfiability.

Theorem 7 holds for a family of strategies, in contrast to Theorem 5 from Bromberget et al., which was only shown for what is here called the minimally backtracking strategy. This family of strategies contains any strategy that preserves the required invariants and is restricted by the minimally backtracking strategy. From Lemma 5 we know that these two requirements are fulfilled by the SCL relation but also by the reasonable, regular, and minimally backtracking strategies. This leads to a more intuitive corollary based on runs.

Corollary 8 (`correct_termination_strategies`). *If an SCL, reasonable SCL, regular SCL, or minimally backtracking SCL run starting from the initial state* $(\epsilon; \{\}; \top)$ *terminates in a state* $S = (\Gamma; U; \mathcal{C})$*, formally any of*

- $(\epsilon; \{\}; \top)\, (\Rightarrow^{N,\beta}_{SCL})^* \, S \ \wedge \ (\nexists S'.\, S \Rightarrow^{N,\beta}_{SCL} S')$,
- $(\epsilon; \{\}; \top)\, (\Rightarrow^{N,\beta}_{Rea\text{-}SCL})^* \, S \ \wedge \ (\nexists S'.\, S \Rightarrow^{N,\beta}_{Rea\text{-}SCL} S')$,
- $(\epsilon; \{\}; \top)\, (\Rightarrow^{N,\beta}_{Reg\text{-}SCL})^* \, S \ \wedge \ (\nexists S'.\, S \Rightarrow^{N,\beta}_{Reg\text{-}SCL} S')$, *or*
- $(\epsilon; \{\}; \top)\, (\Rightarrow^{N,\beta}_{Min\text{-}Bac\text{-}SCL})^* \, S \ \wedge \ (\nexists S'.\, S \Rightarrow^{N,\beta}_{Min\text{-}Bac\text{-}SCL} S')$,

then the conclusion of Theorem 7 holds.

Note that each strategy is used with positive polarity in the "run" hypothesis and negative polarity in the "no-more-step" hypothesis. For this reason, it is impossible to provide a corollary with a single requirement to restrict or be restricted by any known strategy.

Traditional saturation-based calculi for first-order logic, e.g. Resolution and Superposition, can learn redundant clauses and thus their implementations require costly checks for non-redundancy. SCL(FOL) learns only non-redundant clauses. Thus, an implementation would not need to check for (forward) non-redundancy. We first repeat the definition of *standard redundancy* as found in [18].

Definition 9. *A clause* C *is redundant w.r.t. a set of clauses* N *and a strict order on clauses* $\prec$ *if* $(\forall C' \in gnd(C).\ \{D' \in gnd(N) \mid D' \prec C'\} \models_{\mathcal{G}} C')$.

We first prove non-redundancy w.r.t. a trail-induced dynamic order and then lift this result to non-redundancy w.r.t. a static order.

Definition 10. *A trail* Γ *induces a well-founded, strict partial order* $\prec^{\Gamma}$*, total on all atoms in* Γ*'s literals. Assuming* Γ *has the form* $L_n^*, \ldots, L_2^*, L_1^*, L_0^*$ *for all* $* \in \{\dagger, (D, \gamma_D)$ *for some* D *and* $\gamma_D\}$*, we have the following ordering.*

$$atom(L_n) \prec^{\Gamma} \cdots \prec^{\Gamma} atom(L_2) \prec^{\Gamma} atom(L_1) \prec^{\Gamma} atom(L_0)$$

In other words, "older" elements on the left are smaller than "newer" elements on the right. Formally:

$$t_1 \prec^{\Gamma} t_2 \longleftrightarrow (\exists i < |\Gamma|.\ \exists j < i.\ t_1 = atom(lit(\Gamma[i])) \wedge t_2 = atom(lit(\Gamma[j])))$$

Compared to Bromberger et al., the trail-induced order is defined on atoms instead of literals and non-redundancy is proven for any lifting to literals.

Theorem 11 (`dynamic_non_redundancy_regular_scl`). *Following conflict resolution in a regular run, formally if*

- $(\epsilon; \{\}; \top) \, (\Rightarrow^{N,\beta}_{\mathit{Reg\text{-}SCL}})^* \, (\Gamma; U; \top)$,
- $(\Gamma; U; \top) \Rightarrow^{N,\beta}_{\mathit{Conflict}} S_1$,
- $S_1 \, (\Rightarrow^{N,\beta}_{\mathit{Skip,Factorize,Resolve}})^+ \, S_n$, *and*
- $S_n \Rightarrow^{N,\beta}_{\mathit{Backtrack}} S_{1+n}$,

then neither is the learned clause $C = \mathit{conflict}(S_n)$ generalized by any initial or learned clause, formally $(\nexists D \in N \cup U.\, \exists \sigma.\, D\sigma = C)$, nor is it redundant w.r.t. $N \cup U$ and the order we get by first lifting the trail-induced order $\prec^{\Gamma}$ from atoms to literals and then taking its multiset extension.

Dynamic non-redundancy with respect to the trail-induced order does not by itself release an implementation from performing backward non-redundancy checks, but it is a strong guarantee on the quality of learned clauses. For backward redundancy checks an order needs to be used that encompasses all dynamic trail-induced orders. An order based on a strict multiset relation has this property. So for backward redundancy we can, e.g., delete subsumed clauses.

Corollary 12 (`static_non_subsumption_regular_scl`). *If a regular run starting from the initial state $(\epsilon; \{\}; \top)$ learns a clause C, formally if*

- $(\epsilon; \{\}; \top) \, (\Rightarrow^{N,\beta}_{\mathit{Reg\text{-}SCL}})^* \, (\Gamma; U; (C; \gamma))$ *and*
- $(\Gamma; U; (C; \gamma)) \Rightarrow^{N,\beta}_{\mathit{Backtrack}} S$,

then C is not subsumed by any of the initial or learned clauses, formally
$\nexists D \in N \cup U.\, \exists \sigma.\, D\sigma \subseteq C$.

All non-redundancy results can be generalized to an arbitrary strategy restricting the regular strategy. We only show one example here and refer the reader to the formalization for the others.

Corollary 13 (`dynamic_non_redundancy_strategy`). *Following conflict resolution in the run of a strategy restricting regular SCL, formally if*

- $(\epsilon; \{\}; \top) \, (\Rightarrow^{N,\beta}_{\mathit{Strategy}})^* \, (\Gamma; U; \top)$,
- $(\Gamma; U; \top) \Rightarrow^{N,\beta}_{\mathit{Conflict}} S_1$,
- $S_1 \, (\Rightarrow^{N,\beta}_{\mathit{Skip,Factorize,Resolve}})^+ \, S_n$,
- $S_n \Rightarrow^{N,\beta}_{\mathit{Backtrack}} S_{1+n}$, *and*
- $\forall S\, S'.\, S \Rightarrow^{N,\beta}_{\mathit{Strategy}} S' \longrightarrow S \Rightarrow^{N,\beta}_{\mathit{Reg\text{-}SCL}} S'$,

then neither is the learned clause generalized by any initial or learned clause, formally $(\nexists D \in N \cup U.\, \exists \sigma.\, D\sigma = C)$, nor is it redundant w.r.t. $N \cup U$ and the order we get by first lifting the trail-induced order $\prec^{\Gamma}$ from atoms to literals and then taking its multiset extension.

During the development of this formalization, we discovered that the original Backtrack rule found in [6] allows to learn a duplicate of the last learned clause, which violates the stated non-redundancy of learned clauses. The original Backtrack rule ensures that the conflict closure is not false under the new trail, but the learned clause could still be in conflict w.r.t. another grounding. Following this conflict, the Backtrack rules would be immediately applicable and would learn the same clause again. This could only happen a finite number of times as backtracking reduces the length of the (finite) trail. As an example, consider the set of clauses $N = \{P(x), Q(y), \neg Q(z) \vee R(z), \neg R(w) \vee S(w), \neg P(v) \vee \neg S(v)\}$, and a big enough β. The following SCL run was valid with the original Backtrack rule. Note that the notation for the trail was shortened to save space.

$$\begin{array}{ll}
 & (\epsilon; \{\}; \top) \\
(\Rightarrow^{N,\beta}_{\text{Decide}})^* & (P(a), Q(a), P(b), Q(b); \{\}; \top) \\
(\Rightarrow^{N,\beta}_{\text{Propagate}})^* & (P(a), Q(a), P(b), Q(b), R(b)^{(R(z);\neg Q(z);z \mapsto b)}, S(b)^{(S(w);\neg R(w);w \mapsto b)}; \{\}; \top) \\
\Rightarrow^{N,\beta}_{\text{Conflict}} & (P(a), Q(a), P(b), Q(b), R(b)^{(R(z);\neg Q(z);z \mapsto b)}, S(b)^{(S(w);\neg R(w);w \mapsto b)}; \{\}; (\neg P(v) \vee \neg S(v); v \mapsto b)) \\
\Rightarrow^{N,\beta}_{\text{Resolve+Skip}} & (P(a), Q(a), P(b), Q(b), R(b)^{(R(z);\neg Q(z);z \mapsto b)}; \{\}; (\neg P(v) \vee \neg R(v); v \mapsto b)) \\
\Rightarrow^{N,\beta}_{\text{Resolve+Skip}} & (P(a), Q(a), P(b), Q(b); \{\}; (\neg P(v) \vee \neg Q(v); v \mapsto b)) \\
\Rightarrow^{N,\beta}_{\text{Backtrack}} & (P(a), Q(a), P(b); \{\neg P(v) \vee \neg Q(v)\}; \top) \\
\Rightarrow^{N,\beta}_{\text{Conflict+Skip}} & (P(a), Q(a); \{\neg P(v) \vee \neg Q(v)\}; (\neg P(v) \vee \neg Q(v); v \mapsto a)) \\
\Rightarrow^{N,\beta}_{\text{Backtrack}} & (P(a); \{\neg P(v) \vee \neg Q(v)\}; \top)
\end{array}$$

This counterexample was only discovered when we failed to prove Theorem 11 in Isabelle. Note that this formalization is based on and was developed simultaneously to Bromberger et al., which originally inherited the Backtrack rule from [10]. The solution, which was promptly integrated into this formalization and Bromberger et al., is for the Backtrack rule to find a position without conflict w.r.t. the learned clause. Note that the original Backtrack rule reaches such a state after having learned the same clause finitely often, which has no effect on the set of learned clauses because sets ignore duplicates. Thus, the original Backtrack rule did not invalidate the other properties of the SCL calculus. This discovery is strong evidence of the usefulness of mechanized formalization for both published work and ongoing research: the Isabelle formalization lead to the discovery of a previously unknown bug and it guided the development of the refinement.

A calculus expressed as a state machine terminates if the transition relation starting from the initial state is well-founded following the arrow direction. We prove well-foundedness of regular SCL in three steps: (1) we first prove well-foundedness of SCL without backtracking, denoted $\Rightarrow^{N,\beta}_{\text{SCL-no-Back}}$; (2) we then prove that a regular run can only learn finitely many clauses; and (3) from these two results we finally prove well-foundedness of regular SCL. Step 1 is novel to the formalization. Prior work in Bromberger et al. focuses exclusively on the Backtrack rule (step 2) in order to prove termination of regular SCL (step 3). Also novel to the formalization are decreasing measuring functions for steps 1 and 2.

Definition 14. *The measuring function $\mathcal{M}_3(N, \beta, S)$ for SCL without backtracking maps a set of initial clauses N, a bounding atom β, and a state S to a 4-tuple. The tuple elements are (1) a boolean identifying whether the state is conflict-free, (2) a (finite) set overapproximating the literals that could be added to the trail, (3) a (finite) list overapproximating the numbers of resolution steps that could be performed at each position in the trail, and (4) the (finite) cardinality of the conflicting clause. Formally:*

$$\mathcal{M}_1(\beta, \Gamma) \;=\; \{L \mid \mathit{atom}(L) \preceq_B \beta\} - \{\mathit{lit}(\mathcal{K}) \mid \mathcal{K} \in \mathit{set}(\Gamma)\}$$

$$\begin{aligned}
\mathcal{M}_2(\epsilon, C) &= \epsilon \\
\mathcal{M}_2((\Gamma, K), C) &= \mathcal{M}_2(\Gamma, C), 0 \\
\mathcal{M}_2((\Gamma, (K\gamma)^{(K;D;\gamma)}), C) &= \mathit{let}\ n = \mathit{count}(C, \mathit{comp}(K\gamma))\ \mathit{in} \\
&\ \mathcal{M}_2(\Gamma, C \vee \mathit{repeat}(n, D\gamma)), n
\end{aligned}$$

$$\begin{aligned}
\mathcal{M}_3(N, \beta, (\Gamma; U; \top)) &= (\mathit{True};\ \mathcal{M}_1(\beta, \Gamma);\ \epsilon;\ 0) \\
\mathcal{M}_3(N, \beta, (\Gamma; U; (C; \gamma))) &= (\mathit{False};\ \{\};\ \mathcal{M}_2(\Gamma, C);\ |C|)
\end{aligned}$$

With this, we can prove termination of SCL without backtracking (step 1).

Theorem 15 (termination_scl_without_back). *SCL without backtracking is well-founded on all states reachable by an SCL-without-backtracking run starting from the initial state, formally on $\{S \mid (\epsilon; \{\}; \top)\ (\Rightarrow^{N,\beta}_{\mathit{SCL\text{-}no\text{-}Back}})^*\ S\}$.*

We now turn to proving termination of regular SCL with backtracking by first defining an appropriate measuring function.

Definition 16. *The measuring function $\mathcal{M}_4(\beta, S)$ for the rule Backtrack maps a bounding atom β and a state S to a finite set of clauses without duplicates. It computes an over-approximation of the set of clauses that could still be learned modulo duplicates. Formally:*

$$\mathcal{M}_4(\beta, S) \;=\; 2^{\{L \mid \mathit{atom}(L) \preceq_B \beta\}} - \{\mathit{set}(C) \mid C \in \mathit{gnd}(\mathit{learned}(S))\}$$

We then prove that it decreases every time we learn a new clause (step 2).

Lemma 17 ($\mathcal{M}$_back_after_regular_backtrack). *Following conflict resolution in a regular run, formally if*

- $(\epsilon; \{\}; \top)\ (\Rightarrow^{N,\beta}_{\mathit{Reg\text{-}SCL}})^*\ (\Gamma; U; \top)$,
- $(\Gamma; U; \top)\ \Rightarrow^{N,\beta}_{\mathit{Conflict}}\ S_1$,
- $S_1\ (\Rightarrow^{N,\beta}_{\mathit{Skip,Factorize,Resolve}})^+\ S_n$, *and*
- $S_n\ \Rightarrow^{N,\beta}_{\mathit{Backtrack}}\ S_{1+n}$, *then*

1. *the ground conflict is distinct from all groundings of initial and learned clauses modulo duplicates, formally $(\exists C\, \gamma.\ \mathit{conflict}(S_n) = (C; \gamma) \wedge \mathit{set}(C\gamma) \notin \{\mathit{set}(D) \mid D \in \mathit{gnd}(N \cup U)\})$, and*

2. *the set of clauses that could potentially be learned strictly diminishes, formally* $\mathcal{M}_4(\beta, S_{1+n}) \subset \mathcal{M}_4(\beta, S_n)$.

Lemma 17 is novel to the formalization. Together with Theorem 15 it allows us to prove termination of regular SCL with backtracking (step 3).

Theorem 18 (`termination_regular_scl`). *Regular SCL is well-founded on all states reachable by a regular-SCL run starting from the initial state, formally on* $\{S \mid (\epsilon; \{\}; \top)\, (\Rightarrow^{N,\beta}_{\textit{Reg-SCL}})^*\, S\}$.

All termination results can be generalized to an arbitrary strategy restricting the regular strategy. We only show one example here and refer the reader to the formalization for the others.

Corollary 19 (`termination_strategy`). *If a strategy restricts regular SCL, formally if* $(\forall S\, S'.\ S \Rightarrow^{N,\beta}_{\textit{Strategy}} S' \longrightarrow S \Rightarrow^{N,\beta}_{\textit{Reg-SCL}} S')$, *then it is well-founded on all states reachable by a run using this strategy and starting from the initial state, formally on* $\{S \mid (\epsilon; \{\}; \top)\, (\Rightarrow^{N,\beta}_{\textit{Strategy}})^*\, S\}$.

All theorems until now were first expressed and proven using invariants and then the versions expressed using runs were derived. However, Theorem 18 posed an interesting problem because its proof requires the backtracking step to have knowledge of the trail when a conflict last occurred. But this information is lost in the SCL state due to the Skip rule shrinking the trail. We did define an invariant that expresses the historical form of the trail and its properties derived from the regular strategy, but it is complex and the added value compared to working directly on a regular run is questionable. For simplicity, we chose not to present this invariant in this paper.

Together, soundness and termination allow us to prove refutational completeness of the regular SCL calculus w.r.t. a fixed bound.

Theorem 20 (`completeness_wrt_bound`). *If the groundings* $\textit{gnd}^{\preceq_B \beta}(N)$ *of the initial clauses* N *are unsatisfiable, then all regular SCL runs starting from the initial state terminate and derive the conflicting clause* $\bot$, *formally*

1. *there is no infinite regular run starting from the initial state, and*
2. $(\forall S.\ (\epsilon; \{\}; \top)\, (\Rightarrow^{N,\beta}_{\textit{Reg-SCL}})^*\, S \wedge (\nexists S'.\ S \Rightarrow^{N,\beta}_{\textit{Reg-SCL}} S') \longrightarrow (\exists \gamma.\ \textit{conflict}(S) = (\bot; \gamma)))$.

Theorem 20 is only defined w.r.t. a bound, but fortunately we can prove that there must always exist an appropriate bound.

Lemma 21 (`ex_bound_if_unsat`). *If the relation* $\prec_B$ *is a well-founded, strict order, total on ground atoms and the groundings* $\textit{gnd}(N)$ *of the initial clauses* N *are unsatisfiable, then there exists a bound* β *such that the groundings* $\textit{gnd}^{\preceq_B \beta}(N)$ *are unsatisfiable.*

Note that while Lemma 21 proves the existence of an appropriate bound, it provides no constructive way of finding one. What one can do is follow along Theorem 6 and iteratively increase a heuristically chosen bound until an appropriate one is found; if the set of initial clauses is unsatisfiable, this will terminate.

Isabelle Technicalities. Lemma 21's hypothesis that $\prec_B$ is a well-founded, total, strict order cannot be expressed as a theorem-local hypothesis. The reason is that the compactness theorem for clausal first-order logic requires terms to be an instance of the `wellorder` type class, which is not the case in the `scl_fol_calculus` locale, where the assumptions on the $\prec_B$ relation are kept minimal. Because Isabelle does not allow to instantiate a type class with a concrete type inside a locale or theorem, we define a new locale that extends `scl_fol_calculus` and adds a type class requirement on the first-order term constants. This enables the type-class system to automatically instantiate the `wellorder` type class for terms using the previously registered Knuth-Bendix order. We then instantiate the $\prec_B$ relation of `scl_fol_calculus` with the Knuth-Bendix order. This type class and locale gymnastic could be avoided if the formalization of the compactness theorem was refactored to offer a predicate-based version alongside the existing type-class-based version.

4 Conclusion

We generalized and formalized the SCL(FOL) calculus in Isabelle/HOL. The main results are formal proofs of soundness, non-redundancy of learned clauses, termination, and refutational completeness. Because the formalization was performed simultaneously to Bromberger et al., they could benefit from each other. A mechanized formalization must consider low-level details, but it is also the opportunity to identify the most import aspects of the theory and abstract over details needed in the context of an actual implementation. For example, we abstracted from the level of a state to define the Backtracking rule and replaced it with an abstract specification of the result. A level was used in all pen-an-paper presentations of the calculus in order to have a constructive way of going back to the maximal trail where the learned clause propagates. The abstraction supports investigation of several Backtrack rule versions and to base the soundness result on a version with a minimal requirement, i.e., the learned clause is no longer false with respect to the trail.

The formalization did uncover a small bug in the calculus, but also showed that its effect was very localized and naturally lead to a solution. Another benefit of the formalization is how much it supports refactoring and exploratory experimentation. When making a change to a definition or a conjecture, Isabelle immediately and exhaustively points to the parts that need to be adapted. Very often, proofs can automatically be adapted using proof automation tools such as Sledgehammer. This was invaluable to quickly try out ideas or change subtle parts of the calculus. One such example is in the Resolve rule, where the formalization first used substitution composition as found in the original calculus and latter replaced it by an abstract specification of merged grounding. This idea came from a private discussion sketching an eventual C implementation where it became clear that substitution composition would be a costly operation. We then introduced the abstract specification of merged grounding and fixed the formalization by following the mistakes reported by Isabelle.

Acknowledgments. We thank Sophie Tourret and Jasmin Blanchette for many fruitful discussions. We thank the anonymous reviewers for their constructive feedback on this paper.

References

1. Balarin, C.: Locales: a module system for mathematical theories. J. Autom. Reason. **52**(2), 123–153 (2014). https://doi.org/10.1007/s10817-013-9284-7
2. Blanchette, J.C.: Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In: Mahboubi, A., Myreen, M.O. (eds.) Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, 14–15 January 2019, pp. 1–13. ACM (2019). https://doi.org/10.1145/3293880.3294087
3. Blanchette, J.C., Fleury, M., Lammich, P., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. J. Autom. Reason. **61**(1–4), 333–365 (2018). https://doi.org/10.1007/s10817-018-9455-7
4. Blanchette, J.C., Fleury, M., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 25–44. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_4
5. Blanchette, J.C., Tourret, S.: Extensions to the comprehensive framework for saturation theorem proving. Archive of Formal Proofs (2020). https://isa-afp.org/entries/Saturation_Framework_Extensions.html. Formal proof development
6. Bromberger, M., Fiori, A., Weidenbach, C.: Deciding the Bernays-Schoenfinkel fragment over bounded difference constraints by simple clause learning over theories. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) VMCAI 2021. LNCS, vol. 12597, pp. 511–533. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67067-2_23
7. Bromberger, M., Schwarz, S., Weidenbach, C.: Exploring partial models with SCL. In: Konev, B., Schon, C., Steen, A. (eds.) Proceedings of the Workshop on Practical Aspects of Automated Reasoning Co-located with the 11th International Joint Conference on Automated Reasoning (FLoC/IJCAR 2022), Haifa, Israel, 11–12, August 2022. CEUR Workshop Proceedings, vol. 3201. CEUR-WS.org (2022). http://ceur-ws.org/Vol-3201/paper5.pdf
8. Bromberger, M., Schwarz, S., Weidenbach, C.: SCL(FOL) revisited (2023). https://doi.org/10.48550/ARXIV.2302.05954
9. Desharnais, M.: A formalization of the SCL(FOL) calculus: Simple clause learning for first-order logic. Archive of Formal Proofs (2023). https://isa-afp.org/entries/Simple_Clause_Learning.html. Formal proof development
10. Fiori, A., Weidenbach, C.: SCL clause learning from simple models. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 233–249. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_14
11. Huet, G., Oppen, D.C.: Equations and rewrite rules: a survey. Formal Language Theory, pp. 349–405 (1980)
12. Leidinger, H., Weidenbach, C.: SCL(EQ): SCL for first-order logic with equality. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 228–247. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_14
13. Paulson, L.C., Blanchette, J.C.: Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G.,

Schulz, S., Ternovska, E. (eds.) The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, 9 October 2011. EPiC Series in Computing, vol. 2, pp. 1–11. EasyChair (2010). https://doi.org/10.29007/36dt

14. Schlichtkrull, A., Blanchette, J.C., Traytel, D., Waldmann, U.: Formalization of Bachmair and Ganzinger's ordered resolution prover. Archive of Formal Proofs (2018). https://isa-afp.org/entries/Ordered_Resolution_Prover.html. Formal proof development
15. Schlichtkrull, A., Blanchette, J.C., Traytel, D., Waldmann, U.: Formalizing Bachmair and Ganzinger's ordered resolution prover. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 89–107. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_7
16. Schlichtkrull, A., Blanchette, J.C., Traytel, D., Waldmann, U.: Formalizing Bachmair and Ganzinger's ordered resolution prover. J. Autom. Reason. **64**(7), 1169–1195 (2020). https://doi.org/10.1007/s10817-020-09561-0
17. Sternagel, C., Thiemann, R.: First-order terms. Archive of Formal Proofs (2018). https://isa-afp.org/entries/First_Order_Terms.html. Formal proof development
18. Waldmann, U., Tourret, S., Robillard, S., Blanchette, J.C.: A comprehensive framework for saturation theorem proving. J. Autom. Reason. **66**(4), 499–539 (2022). https://doi.org/10.1007/s10817-022-09621-7
19. Wenzel, M.: Isabelle/Isar–a generic framework for human-readable proof documents. In: Matuszewski, R., Zalewska, A. (eds.) From Insight to Proof: Festschrift in Honour of Andrzej Trybulec, Studies in Logic, Grammar, and Rhetoric, vol. 10, no. 23. University of Białystok (2007)

SCL(FOL) Can Simulate Non-Redundant Superposition Clause Learning

Martin Bromberger[1], Chaahat Jain[1,2], and Christoph Weidenbach[1](✉)

[1] Max Planck Institute for Informatics, Saarbrücken, Germany
{mbromber,cjain,weidenbach}@mpi-inf.mpg.de
[2] Graduate School of Computer Science, Saarbrücken, Germany

Abstract. We show that SCL(FOL) can simulate the derivation of non-redundant clauses by superposition for first-order logic without equality. Superposition-based reasoning is performed with respect to a fixed reduction ordering. The completeness proof of superposition relies on the grounding of the clause set. It builds a ground partial model according to the fixed ordering, where minimal false ground instances of clauses then trigger non-redundant superposition inferences. We define a respective strategy for the SCL calculus such that clauses learned by SCL and superposition inferences coincide. From this perspective the SCL calculus can be viewed as a generalization of the superposition calculus.

Keywords: first-order reasoning · superposition · SCL · non-redundant clause learning

1 Introduction

Superposition [1,2,18] is currently considered as the prime calculus for first-order logic reasoning where all leading first-order theorem provers implement a variant thereof [14,16,20,22]. More recently, the family of SCL calculi (Clause Learning from Simple Models, or just Simple Clause Learning) [4,8,9,11,17] was introduced. There are first experimental results [3] available, and first steps towards an overall implementation [5,7].

The main differences between superposition and SCL for first-order logic without equality are: (i) superposition assumes a fixed ordering on literals whereas the ordering in SCL is dynamic and evolves out of the satisfiability of clauses, (ii) superposition performs single superposition left and factoring inferences whereas SCL typically performs several such inferences to derive a single learned clause, (iii) the superposition model operator is not effective on the non-ground clause level whereas the SCL model assumption is effective. For first-order logic without equality superposition reduces to ordered resolution combined with the powerful superposition redundancy criterion. Our simulation result cannot be one-to-one because an SCL learned clause is typically generated by several superposition inferences and superposition factoring inferences are performed by SCL only in the context of resolution inferences. The simulation result considers the

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 134–152, 2023.
https://doi.org/10.1007/978-3-031-38499-8_8

ground case, where the superposition strategy used in the completeness proof only triggers non-redundant inferences [1]. We call this strategy SUP-MO, Definition 5. Overall first-order superposition completeness is then obtained by a lifting argument to the non-ground clause level. We actually show that a superposition refutation of some ground clause set can be simulated by an SCL refutation on the same clause set, such that they coincide on all superposition left (ordered resolution) inferences. For the superposition calculus we refer to [1] and for SCL to [9] where all main properties of both calculi have meanwhile been verified inside the Isabelle framework [10,19,21].

For example, consider a superposition refutation of the simple ground clause set

$$N^0_{\text{SUP}} = \{(C_1)\ P(a) \vee P(a), \quad (C_2)\ \neg P(a) \vee Q(b), \quad (C_3)\ \neg Q(b)\}$$

with respect to a KBO [13], where all symbols have weight one, and precedence $a \prec b \prec P \prec Q$. Superposition generates only non-redundant clauses. Then with respect to the usual superposition ordering extension to literals and clauses we get $(C_1) \prec_{\text{KBO-SUP}} (C_2) \prec_{\text{KBO-SUP}} (C_3)$ and the superposition model operator produces the Herbrand model $N^0_{\text{SUP},\mathcal{I}} = \emptyset$. Now clause (C_1) is the minimal false clause, triggering a factoring inference resulting in (C_4) $P(a)$ and clause set $N^1_{\text{SUP}} = N^0_{\text{SUP}} \cup \{(C_4)\ P(a)\}$. The clause $P(a)$ cannot be derived by SCL because factoring is only preformed in the context of resolution inferences. Now (C_4) is the smallest clause in N^1_{SUP} and the superposition model operator produces $N^1_{\text{SUP},\mathcal{I}} = \{P(a), Q(b)\}$ with minimal false clause (C_3). A superposition left inference between (C_3) and (C_2) generates (C_5) $\neg P(a)$ and $N^2_{\text{SUP}} = N^1_{\text{SUP}} \cup \{(C_5)\ \neg P(a)\}$. The generation of $\neg P(a)$ can now be simulated by SCL by constructing the SCL trail $[P(a)^1 Q(b)^{\{\neg P(a) \vee Q(b)\}}]$ out of $N^0_{\text{SUP}} = N^0_{\text{SCL}}$ leading to the learned clause (C_5) $\neg P(a)$ and respective clause set $N^2_{\text{SCL}} = N^0_{\text{SCL}} \cup \{(C_5)\ \neg P(a)\}$. Note that $P(a)$ could have also been propagated, see Sect. 2 rule Propagate, but this would eventually not lead to the learned clause (C_5) $\neg P(a)$ but $\bot$. Finally, the superposition model operator produces $N^2_{\text{SUP},\mathcal{I}} = \{P(a), Q(b)\}$ with minimal false clause (C_5) and infers $\bot$. The SCL simulation generates the trail $[P(a)^{\{P(a)\}}]$ and then learns $\bot$ as well out of a conflict with (C_5). Note that this SCL trail is based on a factoring of (C_1) to $P(a)$ that was the explicit first step of the superposition refutation. Recall that by using an exhaustive propagation strategy, SCL would start with the trail $[P(a)^{P(a)} Q(b)^{\{\neg P(a) \vee Q(b)\}}]$ and immediately derive $\bot$. Exhaustive propagation is not a good strategy in general, because first-order logic clauses may enable infinitely many propagations. Even together with the typical SCL restriction to finitely many ground instances, there are exponentially many propagations possible, in general. Therefore, the *regular* strategy defined in [9] does not require exhaustive propagation, but guarantees non-redundant clause learning. The SCL-SUP strategy, Definition 8, and Definition 10, simulating superposition SUP-MO runs is also a regular strategy, Lemma 17.

The paper is now organized as follows. After repetition of the needed concepts of SCL and superposition, Sect. 2, the simulation result is contained in Sect. 3. We show that any superposition refutation of a ground clause set producing only non-redundant inferences through the SUP-MO strategy, can be simulated via the SCL-SUP strategy. Based on the 14 simulation invariants of Definition 7, we show the invariants by an inductive argument on the length of the superposition refutation, starting from the initial state, Lemma 13, for intermediate superposition inference steps Lemma 14, until the final refutation Lemma 15, and Lemma 16. For the simulation we do not consider selection in superposition inferences in favor of a less complicated presentation. The paper ends with a discussion of the obtained results. A full version of the paper including all proofs is available on arxiv [6].

2 Preliminaries

We assume a first-order language without equality where N denotes a clause set; C, D denote clauses; L, K, H denote literals; A, B denote atoms; P, Q, R denote predicates; t, s terms; f, g, h function symbols; a, b, c constants; and x, y, z variables. Atoms, literals, clauses and clause sets are considered as usual, where in particular clauses are identified both with their disjunction and multiset of literals [9]. The complement of a literal is denoted by the function comp. The function $\text{atom}(L)$ denotes the atomic part of a literal. Semantic entailment $\models$ is defined as usual where variables in clauses are assumed to be universally quantified. Substitutions σ, τ are total mappings from variables to terms, where $\text{dom}(\sigma) := \{x \mid x\sigma \neq x\}$ is finite and $\text{codom}(\sigma) := \{t \mid x\sigma = t, x \in \text{dom}(\sigma)\}$. Their application is extended to literals, clauses, and sets of such objects in the usual way. A term, atom, clause, or a set of these objects is *ground* if it does not contain any variable. A substitution σ is *ground* if $\text{codom}(\sigma)$ is ground. A substitution σ is *grounding* for a term t, literal L, clause C if $t\sigma$, $L\sigma$, $C\sigma$ is ground, respectively. The function mgu denotes the *most general unifier* of two terms, atoms, literals. We assume that any mgu of two terms or literals does not introduce any fresh variables and is idempotent. A *closure* is denoted as $C \cdot \sigma$ and is a pair of a clause C and a substitution σ that is grounding for C. The function ground returns the set of all ground instances of a literal, clause, or clause set with respect to the signature of the respective clause set.

A *(partial) model* M for a clause set N is a satisfiable set of ground literals. A ground clause C is true in M, denoted $M \models C$, if $C \cap M \neq \emptyset$, and false otherwise. A ground clause set N is true in M, denoted $M \models N$ if all clauses from N are true in M. A *(partial) Herbrand model* I for a clause set N is a set of ground atoms. A ground clause C is true in I, denoted $I \models_H C$, if there is an atom $A \in C$ such that $A \in I$, or there is a negative literal $\neg A \in C$ such that $A \notin I$, and false otherwise. A ground clause set N entails a ground clause C, denoted $N \models C$, if $M \models C$ implies $M \models \{C\}$ for all models M.

We identify sets and sequences whenever appropriate. However, the trail of an SCL run is always a sequence of ground literals.

Let $\prec$ denote a well-founded, total, strict ordering on ground literals. This ordering is then lifted to clauses and clause sets by its respective multiset extension. We overload $\prec$ for literals, clauses, clause sets if the meaning is clear from the context. The ordering is lifted to the non-ground case via instantiation: we define $C \prec D$ if for all grounding substitutions σ it holds $C\sigma \prec D\sigma$. We define $\preceq$ as the reflexive closure of $\prec$ and $N^{\preceq C} := \{D \mid D \in N \text{ and } D \preceq C\}$.

Definition 1 (Clause Redundancy). *A ground clause C is* redundant *with respect to a ground clause set N and an order $\prec$ if $N^{\preceq C} \models C$. A clause C is* redundant *with respect to a clause set N and an order $\prec$ if for all $C' \in$* ground(C) *it holds that C' is redundant with respect to* ground(N)*.*

Let $\prec_B$ denote a well-founded, total, strict ordering on ground atoms such that for any ground atom A there are only finitely many ground atoms B with $B \prec_B A$. For example, an instance of such an ordering could be KBO without zero-weight symbols. (Note that LPO does not satisfy the last condition of a $\prec_B$ ordering although it is a well-founded, total, strict ordering.) The ordering $\prec_B$ is lifted to literals by comparing the respective atoms and if the atoms of two literals are the same, then the negative version of the literal is larger than the positive version. It is lifted to clauses by a multiset extension.

The SCL(FOL) Calculus: The inference rules of SCL(FOL) [9] are represented by an abstract rewrite system. They operate on a problem state, a six-tuple $(\Gamma; N; U; \beta; k; D)$ where Γ is a sequence of annotated ground literals, the *trail*; N and U are the sets of *initial* and *learned* clauses; β is a ground literal limiting the size of the trail; k counts the number of decisions; and D is either $\top$, $\bot$ or a clause closure $C \cdot \sigma$ such that $C\sigma$ is ground and false in Γ. Literals in Γ are either annotated with a number, also called a level; i.e., they have the form L^k meaning that L is the k-th guessed decision literal, or they are annotated with a closure that propagated the literal to become true. A ground literal L is of *level i* with respect to a problem state $(\Gamma; N; U; \beta; k; D)$ if L or comp(L) occurs in Γ and the first decision literal left from L (comp(L)) in Γ, including L, is annotated with i. If there is no such decision literal then its level is zero. A ground clause D is of *level i* with respect to a problem state $(\Gamma; N; U; \beta; k; D)$ if i is the maximal level of a literal in D. The level of the empty clause $\bot$ is 0. Recall D is a non-empty closure or $\top$ or $\bot$. Similarly, a trail Γ is of level i if the maximal literal in Γ is of level i.

A literal/atom L/A is *undefined* in Γ if neither L/A nor comp(L)/comp(A) occur in Γ. The start state of SCL is $(\epsilon; N; \emptyset; \beta; 0; \top)$ for some initial clause set N and bound β. The below rules are exactly the rules from [9] and serve as a reference for our simulation proof in Sect. 3.

Propagate $(\Gamma; N; U; \beta; k; \top) \Rightarrow_{\text{SCL}} (\Gamma, L\sigma^{(C_0 \vee L)\delta\cdot\sigma}; N; U; \beta; k; \top)$

provided $C \vee L \in (N \cup U)$, $C = C_0 \vee C_1$, $C_1\sigma = L\sigma \vee \cdots \vee L\sigma$, $C_0\sigma$ does not contain $L\sigma$, δ is the mgu of the literals in C_1 and L, $(C \vee L)\sigma$ is ground, $(C \vee L)\sigma \prec_\beta \{\beta\}$, $C_0\sigma$ is false under Γ, and $L\sigma$ is undefined in Γ.

Decide $(\Gamma; N; U; \beta; k; \top) \Rightarrow_{\text{SCL}} (\Gamma, L\sigma^{k+1}; N; U; \beta; k+1; \top)$

provided atom(L) occurs C for a $C \in (N \cup U)$, $L\sigma$ is a ground literal undefined in Γ, and $L\sigma \prec_\beta \beta$.

Conflict $(\Gamma; N; U; \beta; k; \top) \Rightarrow_{\text{SCL}} (\Gamma; N; U; \beta; k; D \cdot \sigma)$

provided $D \in (N \cup U)$, $D\sigma$ false in Γ for a grounding substitution σ.

Skip $(\Gamma, L; N; U; \beta; k; D \cdot \sigma) \Rightarrow_{\text{SCL}} (\Gamma; N; U; \beta; k - i; D \cdot \sigma)$

provided comp(L) does not occur in $D\sigma$, if L is a decision literal then $i = 1$, otherwise $i = 0$.

Factorize $(\varGamma; N; U; \beta; k; (D \lor L \lor L') \cdot \sigma) \Rightarrow_{\text{SCL}} (\varGamma; N; U; \beta; k; (D \lor L)\eta \cdot \sigma)$ provided $L\sigma = L'\sigma$, $\eta = \text{mgu}(L, L')$.

Resolve $(\varGamma, L\delta^{(C \lor L)\cdot\delta}; N; U; \beta; k; (D \lor L') \cdot \sigma)$
$\Rightarrow_{\text{SCL}}$ $(\varGamma, L\delta^{(C \lor L)\cdot\delta}; N; U; \beta; k; (D \lor C)\eta \cdot \sigma\delta)$
provided $L\delta = \text{comp}(L'\sigma)$, $\eta = \text{mgu}(L, \text{comp}(L'))$.

Backtrack $(\varGamma_0, K, \varGamma_1, \text{comp}(L\sigma)^k; N; U; \beta; k; (D \lor L) \cdot \sigma)$
$\Rightarrow_{\text{SCL}}$ $(\varGamma_0; N; U \cup \{D \lor L\}; \beta; j; \top)$
provided $D\sigma$ is of level $i' < k$, and $\varGamma_0, K$ is the minimal trail subsequence such that there is a grounding substitution τ with $(D \lor L)\tau$ is false in $\varGamma_0, K$ but not in $\varGamma_0$, and $\varGamma_0$ is of level j.

A sequence of rule applications of a particular calculus is called a *run* of the calculus. A *strategy* for a calculus restricts the set of runs we actually allow by imposing further conditions on the allowed rule applications.

Definition 2 (SCL Runs). *A sequence of SCL rule applications is called a* reasonable run *if the rule Decide does not enable an immediate application of rule Conflict. A sequence of SCL rule applications is called a* regular run *if it is a reasonable run and the rule Conflict has precedence over all other rules.*

All regular SCL runs are sound, only derive non-redundant clauses, always terminate, and SCL with a regular strategy is refutationally complete (for first-order logic without equality) [9].

The Superposition Calculus: Superposition [1,2,18] is a calculus for first-order logic reasoning that also infers/learns new clauses like SCL. In contrast to SCL, it does these inferences based on a static ordering $\prec$ and, at the level of inference rules, independent of a partial model. A permissible ordering $\prec$ for the superposition calculus is always a well-founded, total, strict ordering on ground literals. This ordering is then lifted to clauses and clause sets by its respective multiset extension. A problem state in the superposition calculus is just a set N of clauses. The start state the initial clause set. Due to the restriction to first-order logic without equality, the most basic version of the superposition calculus consists just of the following two rules (without selection):

Superposition Left $(N \uplus \{C_1 \lor P(t_1, \ldots, t_n), C_2 \lor \neg P(s_1, \ldots, s_n)\}) \Rightarrow_{\text{SUP}}$ $(N \cup \{C_1 \lor P(t_1, \ldots, t_n), C_2 \lor \neg P(s_1, \ldots, s_n)\} \cup \{(C_1 \lor C_2)\sigma\})$
where (i) $P(t_1, \ldots, t_n)\sigma$ is strictly maximal in $(C_1 \lor P(t_1, \ldots, t_n))\sigma$ (ii) $\neg P(s_1, \ldots, s_n)\sigma$ is maximal, (iii) σ is the mgu of $P(t_1, \ldots, t_n)$ and $P(s_1, \ldots, s_n)$.

Factoring $(N \uplus \{C \lor P(t_1, \ldots, t_n) \lor P(s_1, \ldots, s_n)\}) \Rightarrow_{\text{SUP}}$ $(N \cup \{C \lor P(t_1, \ldots, t_n) \lor P(s_1, \ldots, s_n)\} \cup \{(C \lor P(t_1, \ldots, t_n))\sigma\})$
where (i) $P(t_1, \ldots, t_n)\sigma$ is maximal in $(C \lor P(t_1, \ldots, t_n) \lor P(s_1, \ldots, s_n))\sigma$ (ii) σ is the mgu of $P(t_1, \ldots, t_n)$ and $P(s_1, \ldots, s_n)$.

Let sfac(C) represent a clause obtained by exhaustively applying superposition Factoring on C. Recall, that superposition Factoring only applies to maximal positive literals. Let sfac(N) represent the clause set N after every clause has been exhaustively factorized by Superposition Factorization.

Although the superposition calculus itself is independent of a partial model and may learn non-redundant clauses, the completeness proof of superposition in [1] is based on a strategy that builds ground partial models according to the fixed ordering $\prec$, where minimal false ground instances of clauses then trigger non-redundant superposition inferences. Note that the completeness proof relies on a grounding of the clause set that may lead to infinitely many clauses. However, the strategy from the completeness proof can also be seen as a superposition strategy for an initial clause set, where all clauses are already ground. On ground, finite clause sets, superposition restricted to the strategy only infers non-redundant clauses, always terminates, and is complete. The partial model needed in each step of the strategy is constructed according to the following model operator:

Definition 3 (Superposition Model Operator). *Let N be a set of ground clauses. Then N_I is the Herbrand model according to the superposition model operator for clause set N and it is constructed recursively over the partial Herbrand models N_C for all $C \in N$:*

$$N_C = \bigcup_{D \prec C} \delta_D \qquad\qquad N_I = \bigcup_{C \in N} \delta_C$$

$$\delta_D = \begin{cases} \{B\} & \textit{if } D = D' \vee B,\ B \textit{ strictly maximal},\ N_D \not\models_H D \\ \emptyset & \textit{otherwise} \end{cases}$$

We say that a clause C is productive *(wrt. the model construction of a clause set N) if $\delta_C \neq \emptyset$. We say that a clause C* produces *an atom B (wrt. the model construction of a clause set N) if $\delta_C = \{B\}$.*

After constructing the model N_I for a clause set N, the strategy selects the smallest clause in N that is false in N_I. The strategy then selects a fitting inference rule based on the reason why the clause is false in N_I. The newly inferred clause either changes the model in the next step or changes the smallest clause that is false. This is the strategy used in the superposition completeness proof [1].

Definition 4 (Minimal False Clause). *The minimal false clause $C \in N$ is the smallest clause in N according to $\prec$ such that $N_C \cup \delta_C \not\models_H C$.*

Definition 5 (Superposition Model-Operator Strategy: SUP-MO). *The* superposition model-operator strategy *is defined over the minimal false clause with regards to the current clause set N. The strategy can encounter the following cases:*

(1) *N has no minimal false clause. Then N is satisfied by N_I and we can stop the superposition run.*

(2) *The minimal false clause in N is $\bot$. Then N is unsatisfiable, which means we can also stop the superposition run.*

(3) *C is the minimal false clause in N, and it has a maximal literal L that is negative. Then there must be a clause $D \in N$ with $D \prec C$, a strictly maximal literal* $\mathrm{comp}(L)$*, and $\delta_D = \{\mathrm{comp}(L)\}$. In this case, the strategy applies as its next step Superposition Left to C and D.*

(4) *C is the minimal false clause in N, and it has a maximal literal L that is positive. Then L is not strictly maximal in C and the strategy applies Factoring to C.*

The first two cases of the SUP-MO strategy also describe its final states according to [1]. In all other states there is always exactly one rule applicable according to the SUP-MO strategy, which also means that SUP-MO is never stuck.

Lemma 6 (SUP-MO Applicability). *Let N be a set of ground clauses. If N has a minimal false clause $C \neq \bot$, then there exists exactly one rule applicable to N according to the SUP-MO strategy.*

3 SCL Simulates Superposition

In general, it is not possible to simulate all inferences of the superposition calculus with SCL because SCL only learns/infers non-redundant clauses, whereas syntactic superposition inferences have no such guarantees. Moreover, the inferences by SCL are all based on conflicts according to a partial model driven by the satisfiability of clause instances, whereas the inferences by superposition are based on a static ordering $\prec$. We can mitigate these differences by restricting superposition with the SUP-MO strategy because SUP-MO has non-redundancy guarantees and it infers new clauses based on minimal false clauses with respect to a ground partial model.

Let N^0 be a set of ground clauses, totally ordered by a superposition reduction ordering $\prec$. Let N^i (for $i > 0$) be the result of i steps of the superposition calculus applied to N^0 according to the SUP-MO strategy, i.e., $N^0 \Rightarrow_{\text{SUP-MO}} N^1 \Rightarrow_{\text{SUP-MO}} \dots \Rightarrow_{\text{SUP-MO}} N^i$. Again, all N^i are sets of ground clauses, totally ordered by a superposition reduction ordering $\prec$. The SCL strategy SCL-SUP that simulates superposition restricted to SUP-MO runs is defined inductively on the clause ordering $\prec$. To guide and to prove the correctness of our simulation, we assign to each SCL state and every clause some additional information. For this purpose, every SCL state is annotated with a triple (i, C, γ), where i is an integer that states that the SCL state simulates the superposition state N^i, C is the last clause that was used as a decision aid by the strategy, γ is a function such that $\gamma(C) = \mathrm{sfac}(C)$ if $\mathrm{sfac}(C) \in N^i$ and $\gamma(C) = C$ otherwise, the SCL state also simulates the model construction for N^i upto $N^i_{C'} \cup \delta_{C'}$, where $C' = \gamma(C)$. The annotated states are written $(\Gamma; N^0; U; \beta; k; E)_{(i,C,\gamma)}$. The overall start state is then $(\epsilon; N^0; \emptyset; \beta; 0; \top)_{(0,\bot,\gamma)}$, where we assume β large enough so $A \prec_\beta \beta$ for all $A \in \mathrm{atom}(N^0)$, $\bot \notin N^0$, and $\gamma(C) = \mathrm{sfac}(C)$ if $\mathrm{sfac}(C) \in N^0$ and $\gamma(C) = C$ otherwise. We will later see that the annotated integer is not relevant for the actual choice of SCL rules by the SCL-SUP strategy but only to prove that the strategy actually simulates superposition. Moreover, we define a new ordering $\prec_\gamma$ based on our superposition ordering $\prec$ and function γ such that $C \prec_\gamma D$ if $\gamma(C) \prec \gamma(D)$.

Definition 7 (State Simulation). *Let $(\Gamma; N^0; U; \beta; k; E)_{(i,D,\gamma)}$ be an SCL state for the input clauses N^0. Let L be the maximal literal in D if $D \neq \bot$ and the minimal literal according to $\prec$ otherwise. Let $N^0 \Rightarrow_{SUP\text{-}MO} N^1 \Rightarrow_{SUP\text{-}MO} \ldots \Rightarrow_{SUP\text{-}MO} N^i$ be the superposition run following the SUP-MO strategy starting from the input clause set N^0. Let $D' = \gamma(D)$. Then we say that the SCL state $(\Gamma; N^0; U; \beta; k; E)_{(i,D,\gamma)}$* simulates *$N^i$ and the model construction upto $N^i_{D'} \cup \delta_{D'}$ if*

- **(i)** $\operatorname{atom}(N^0) = \operatorname{atom}(N^i) = \operatorname{atom}(N^0 \cup U)$, *$A \prec_\beta \beta$ for all $A \in \operatorname{atom}(N^0)$, and $D \in \{\bot\} \cup N^0 \cup U$*
- **(ii)** $\operatorname{sfac}(N^0 \cup U) \subseteq \operatorname{sfac}(N^i)$ *and $\gamma(C) \in N^i$ for all $C \in N^0 \cup U$ and $\gamma(C) = \operatorname{sfac}(C)$ or $\gamma(C) = C$.*
- **(iii)** *for all $C \in N^i$ there exists a $C' \in N^0 \cup U \cup \{E\}$ such that* $\operatorname{sfac}(C) = \operatorname{sfac}(C')$ *if the maximal literal in C is positive*
- **(iv)** *for all $C \in N^i$ there exists a $C' \in N^0 \cup U \cup \{E\}$ such that $C' \models C$ and $\gamma(C') \preceq C$*
- **(v)** *for all atoms A occurring in N^0: $A \in N_{D'} \cup \delta_{D'}$ iff $A \in \Gamma$*
- **(vi)** *for all atoms A: $\neg A \in \Gamma$ iff $A \prec L$ and $A \notin N_{D'}$*
- **(vii)** *for every literal L in Γ, i.e., $\Gamma = \Gamma', L, \Gamma''$, and all literals L' in Γ',* $\operatorname{atom}(L') \prec \operatorname{atom}(L)$
- **(viii)** *for every atom (= positive literal) B in Γ, i.e., $\Gamma = \Gamma', B, \Gamma''$, there exists $C \in N^0 \cup U$ and a $C' \in N^i$ such that $\gamma(C) = \operatorname{sfac}(C) = \operatorname{sfac}(C') = C'$, and C' produces B, i.e., $\delta_{C'} = \{B\}$*
- **(ix)** *for every clause $C \in N^i$ with $C \preceq \gamma(D)$ that produces an atom B, i.e., $\delta_C = \{B\}$, there exists $C' \in N^0 \cup U$ such that $C = \gamma(C')$ and $C \preceq_\gamma D$.*
- **(x)** *Γ contains only decisions if $E = \top$*
- **(xi)** *$E \notin \{\top, \bot\}$ iff $\Gamma = \Gamma' B^{\operatorname{sfac}(D)}$, Γ' contains only decisions, there exists $E' \in N^i$ where $\gamma(E) = E = E'$ is the minimal false clause in N^i, and $\neg B \in E$*
- **(xii)** *$\Gamma \models C$ for all $C \in N^0 \cup U$ with $C \preceq_\gamma D$*
- **(xiii)** *Conflict is not applicable to $(\Gamma; N^0; U; \beta; k; E)_{(i,D,\gamma)}$.*
- **(xiv)** *$\bot \notin N^0 \cup U$ and $E = \bot$ iff $\Gamma = \epsilon$ and $\bot \in N^i$*

The above invariants can be summarized as follows: (i) All ground atoms encountered are known from the start and the trail bound β is large enough so SCL can Decide/Propagate them. (ii)–(iv) Every initial clause C or inferred clause by SUP-MO must coincide with an initial clause C' or learned clause by SCL; this means on the one hand that for every clause C learned by SCL-SUP, SUP-MO infers a clause C' that is identical up to factoring; on the other hand it means that for every clause C inferred by SUP-MO, SCL-SUP learns a clause C' that entails C (i.e. $C' \models C$) and is at most as large as C wrt. γ. (v)–(ix) The partial model constructed by SCL-SUP and SUP-MO coincide and any atom B in $N_C \cup \delta_C$ produced by clause D has a clause D' on the SCL side that could propagate B and vice versa. (x)–(xiii) Ensure that any Conflict in SCL-SUP corresponds to a minimal false clause and that the trail is always constructed in such a way that the Resolve applications per Conflict call are limited to the maximal literal in the conflict; this property is needed or the next clause that would be learned by SCL no longer coincides with the clauses learned by SUP-MO. (xiv) Describes the final state in case the input clause set is unsatisfiable.

Now that we have defined how an SCL state must look like in order to simulate a superposition state, we define SCL-SUP, the SCL strategy that eventually simulates a SUP-MO run. First, note that not all states visited by SCL-SUP satisfy the invariants of Definition 7. However, the invariants hold again after each so-called *atomic sequence* of SCL-SUP steps. Second, one atomic sequence of SCL-SUP steps may skip over several successive superposition states. The reason is that SCL can and must skip all steps of SUP-MO that occur because the maximal literal in a clause is not strictly maximal, i.e., superposition Factoring steps. SCL performs factoring implicitly in its Propagation rule so SCL never has to explicitly simulate case (4) of Definition 5. Third, definition of the SCL-SUP strategy is split in two parts and each part describes some atomic sequences of SCL-SUP steps.

Definition 8 (SCL Superposition Strategy: SCL-SUP Part 1). *Let $S_0 = (\Gamma; N^0; U; \beta; k; \top)_{(i,C,\gamma)}$ be an SCL state with additional annotations for the strategy. Let D be the next largest clause from C in the ordering $\prec_\gamma$ with respect to the ground clause set $N^0 \cup U$. Let L be the maximal literal of D. Let $[\neg A_1, \neg A_2, \ldots \neg A_n]$ be all negative literals such that for all i we have $A_i \prec L$, all A_i undefined in Γ, A_i occurs in $N^0 \cup U$, and $A_i \prec A_{i+1}$. Let $D' = \gamma(D)$ be in N^i such that* $\mathrm{sfac}(D) = \mathrm{sfac}(D')$. *Let $j_0 + 1$ be the number of occurrences of L in D' and $j = i + j_0$. Then the* SCL Superposition Strategy *(SCL-SUP) performs the following steps to S_0 (possibly without any actual SCL rule applications, just changing the state annotation):*

(1) *First decide all literals $[\neg A_1, \neg A_2, \ldots \neg A_n]$ in order, i.e., $S_0 \Rightarrow^{*\,Decide}_{SCL\text{-}SUP} S_1$, where $S_1 = (\Gamma, \neg A_1^{k+1}, \ldots, \neg A_n^{k+n}; N^0; U; \beta; k+n; \top)_{(i,D,\gamma)}$.*

(2a) *If the maximal literal L in D is positive (i.e., $L = B$), $\Gamma, \neg A_1^{k+1}, \ldots, \neg A_n^{k+n} \not\models D$, and Conflict is not applicable to $S_2 = (\Gamma, \neg A_1^{k+1}, \ldots, \neg A_n^{k+n}, B^{k+n+1}; N^0; U; \beta; k+n+1; \top)_{(j,D,\gamma')}$, then decide B, i.e., $S_1 \Rightarrow^{Decide}_{SCL\text{-}SUP} S_2$, where γ' is the same as γ except that $\gamma'(D) =$* $\mathrm{sfac}(D)$.

(2b) *If the maximal literal L in D is positive (i.e., $L = B$), $\Gamma, \neg A_1^{k+1}, \ldots, \neg A_n^{k+n} \not\models D$, and E is the smallest clause in $N^0 \cup U$ that is false in wrt. $\Gamma, \neg A_1^{k+1}, \ldots, \neg A_n^{k+n}, B^{\mathrm{sfac}(D)}$, then propagate B and apply Conflict to E, i.e., $S_1 \Rightarrow^{Propagate}_{SCL\text{-}SUP} S_2' \Rightarrow^{Conflict}_{SCL\text{-}SUP} S_2$, where $S_2 = (\Gamma, \neg A_1^{k+1}, \ldots, \neg A_n^{k+n}, B^{\mathrm{sfac}(D)}; N^0; U; \beta; k+n; E)_{(j,D,\gamma')}$ and γ' is the same as γ except that $\gamma'(D) =$* $\mathrm{sfac}(D)$.

(2c) *Otherwise, $S_2 = S_1$ and no further rules have to be applied.*

A (potentially empty) sequence of SCL rule applications according to SCL-SUP is called an atomic sequence *of SCL-SUP steps if it starts from a state S_0 and ends in a state S_2 outlined in the cases (2a-c).*

The first part of the strategy simulates the recursive construction of the partial model used in the SUP-MO strategy (see Definition 3). It assumes that the model is already constructed up to the current annotated clause C and extends this model for the next largest clause $D \in (N^0 \cup U)$. To this end, it uses the rule Decide in step (1) to set all atoms A to false that are still undefined but can no longer be produced by any clause greater or equal to D. Next the strategy makes a case distinction. Step (2a) handles the case where D corresponds to a clause D' in the superposition state (modulo some Factoring steps skipped by SCL) that produces atom B; SCL-SUP then adds B to the trail with the rule Decide because producing/adding this atom does not falsify a clause. Step (2b) handles a similar case compared to step (2a); but in this case producing/adding the atom B to the trail results in a minimal false clause E; in order to force a resolution step between clause D and E, SCL-SUP first uses Propagate to add B to the trail and then applies conflict to E. Step (2c) handles the case where D corresponds to a clause D' that will not produce an atom B even modulo some Factoring steps; in this case no further SCL rule applications are necessary as the SUP-MO model will not change. Note that the annotated function γ is needed so the SCL state knows when the superposition state would have applied Factoring to a clause C, which also means that it is now treated as its factorized version $\gamma(C) = \mathrm{sfac}(C)$ in our inductive clause ordering.

Example 9. Let us now further demonstrate the three different cases of the first part of the SCL-SUP strategy with the help of an example. Let N^0 be our initial set of clauses:

$$N^0 = \{(C_1)\ P(a), \quad (C_2)\ \neg P(b) \vee Q(a), \quad (C_3)\ \neg P(a) \vee Q(a) \vee Q(a),$$
$$(C_4)\ P(a) \vee \neg Q(a), \quad (C_5)\ \neg P(a) \vee \neg Q(a)\}$$

We compare the run of SCL-SUP for N^0 with the run of SUP-MO for N^0 to demonstrate that both runs coincide. As superposition ordering, we choose an LPO with precedence $a \prec b \prec P \prec Q$. This means that the atoms are ordered $P(a) \prec P(b) \prec Q(a) \prec Q(b)$ and the clauses in N^0 are ordered $C_1 \prec C_2 \prec C_3 \prec C_4 \prec C_5$. The initial SUP-MO state is simply the clause set N^0 and the initial SCL-SUP state is $(\epsilon, N^0, \emptyset, \beta, 0, \top)_{(0,\bot,\gamma_0)}$, where $\gamma_0(C) = C$ for all clauses C. In the first step of SCL-SUP, SCL-SUP first selects the clause C_1 as its new decision aid because it is the next largest clause in N^0 compared to $\bot$. Then SCL-SUP continues with step (1) of Definition 3. In this step SCL-SUP does nothing because there are no atoms smaller than $P(a)$. Next, SCL-SUP detects that the maximal literal of C_1 is positive, $\epsilon \not\models C_1$, and that the trail $[P(a)^1]$ does not result in a conflict. Therefore, SCL-SUP follows step (2a) of Definition 3 and Decides $P(a)$, which results in the state $([P(a)^1], N^0, \emptyset, \beta, 1, \top)_{(0,C_1,\gamma_0)}$. Meanwhile, SUP-MO starts with constructing a model for N^0 starting with the clause C_1. The result is that C_1 is productive and $\delta_{C_1} = \{P(a)\}$ and $N^0_{C_1} = \emptyset$, which coincides with our new SCL trail.

SCL-SUP considers the clause C_2 as its new decision aid and continues with step (1) of Definition 3. This time there is an atom smaller than the maximal literal of C_2 namely $P(b)$. Therefore, SCL-SUP Decides $\neg P(b)$ in step (1) of Definition 3, which results in $([P(a)^1, \neg P(b)^2], N^0, \emptyset, \beta, 2, \top)_{(0,C_2,\gamma_0)}$. Next, SCL-SUP detects that the maximal literal of C_2 is positive but that $[P(a)^1, \neg P(b)^2] \models C_2$. Therefore, SCL-SUP follows step (2c) of Definition 3 and ends this atomic sequence immediately. SUP-MO continues the model construction for N^0 with the clause C_2. The clause C_2 is not productive because $N^0_{C_2} \models_H C_2$, where $N^0_{C_2} = \delta_{C_1} = \{P(a)\}$ and $\delta_{C_2} = \emptyset$, which again coincides with our new SCL trail as Herbrand models do not explicitly define atoms assigned to false.

SCL-SUP now considers the clause C_3 as its new decision aid and continues with step (1) of Definition 3. In this step SCL-SUP does nothing because all atoms smaller than $Q(a)$ are already assigned. Next, SCL-SUP detects that the maximal literal of C_3 is positive, $[P(a)^1, \neg P(b)^2] \not\models C_3$, and that the clause C_5 is false with respect to the trail $[P(a)^1, \neg P(b)^2, Q(a)^{\mathrm{sfac}(C_3)}]$. Therefore, SCL-SUP follows step (2b) of Definition 3, i.e. it Propagates $P(a)$ and applies Conflict to C_5, resulting in $([P(a)^1, \neg P(b)^2, Q(a)^{\mathrm{sfac}(C_3)}], N^0, \emptyset, \beta, 2, C_3)_{(1,C_2,\gamma_1)}$, where γ_1 is identical to γ_0 except that $\gamma_1(C_3) = \mathrm{sfac}(C_3) = \neg P(a) \vee Q(a)$. Note that SCL-SUP must change the state annotations because the maximal literal in C_3 is not strictly maximal, so SCL-SUP skips and eventually silently performs the Factorization step performed by SUP-MO. Note also that in the changed clause ordering $\prec_{\gamma_1}$ the order of C_2 and C_3 changed, i.e., $C_3 \prec_{\gamma_1} C_2$, which corresponds to $\mathrm{sfac}(C_3) \prec C_2$. Meanwhile, SUP-MO continues the model construction for N^0 with the clause C_3. The clause C_3 is not productive because the maximal literal is not strictly maximal so $\delta_{(3)} = \emptyset$ and $N^0_{C_3} \cup \delta_{C_3} \not\models_H C_3$ so C_3 is the minimal false clause in N^0. SUP-MO resolves this conflict by applying Factoring to C_3, which means SUP-MO infers the clause $C_6 = \mathrm{sfac}(C_3) = \neg P(a) \vee Q(a)$. The new clause order in superposition state $N^1 = N^0 \cup \{C_6\}$ is $C_1 \prec C_6 \prec C_2 \prec C_3 \prec C_4 \prec C_5$, which matches the changed ordering $C_3 \prec_{\gamma_1} C_2$ because $C_6 = \gamma_1(C_3)$. Next, SUP-MO updates its model construction for N^1. The result is that C_1 and C_6 are productive and that $N^1_{C_6} \cup \delta_{C_6} = \{P(a), Q(a)\}$, which matches the current SCL trail. Moreover, if we continue the model construction upto C_5 then no new literals are produced and C_5 also turns into the minimal false clause for N^1.

Definition 10 (SCL Superposition Strategy: SCL-SUP Part 2). *Let $S_0 = (\Gamma, B^{\mathrm{sfac}(C)}; N^0; U; \beta; k; E)_{(i,C,\gamma)}$ be an SCL state with $E \notin \{\top, \bot\}$ and additional annotations for the strategy. Let $L = \neg B$ be the maximal literal of E. Let Γ contain only decision literals. Let all atoms A occurring in $N^0 \cup U$ with $A \prec B$ be defined in Γ following the order $\prec$, i.e., for all A occurring in $N^0 \cup U$ with $A \prec B$ there exist Γ' and Γ'' such that $\Gamma' = \Gamma', L_A, \Gamma''$, $L_A = A$ or $L_A = \neg A$ and all atoms $A' \in N^0 \cup U$ with $A' \prec A$ are defined in Γ'. Let E be contained in N^i. Let j_0 be the number of occurrences of L in E and $j = i + j_0$. Let* $\mathrm{sfac}(C) = C_1 \vee B$ *and $E = E' \vee E''$, where E'' contains all occurrences of L in E. Then the* SCL Superposition Strategy *(SCL-SUP) performs the following steps to S_0:*

(1) *First apply Resolve to E until all occurrences of L are resolved away, i.e., $S_0 \Rightarrow^{*\,\mathit{Resolve}}_{\mathit{SCL\text{-}SUP}} S_1$, where $S_2 = (\Gamma, B^{\mathrm{sfac}(C)}; N^0; U; \beta; k; E_2)_{(j,C,\gamma)}$ and $E_2 = E' \vee C_1 \vee \ldots \vee C_1$.*

(2a) *If $E_2 = \bot$, then we apply Skip until the trail is empty and then stop the SCL run, i.e., $S_2 \Rightarrow^{*\,\mathit{Skip}}_{\mathit{SCL\text{-}SUP}} S_5$, where $S_5 = (\epsilon; N^0; U; \beta; 0; \bot)_{(j,\bot,\gamma)}$.*

(2b) *If $E_2 \neq \bot$, then E_2 has a maximal literal L_1. Next the strategy applies Skip until* comp(L_1) *is the topmost literal on the trail, i.e., $S_2 \Rightarrow^{*\,\mathit{Skip}}_{\mathit{SCL\text{-}SUP}} S_3$, where $S_3 = (\Gamma_0, L_1^{k_1}; N^0; U; \beta; k_1; E_2)_{(j,C,\gamma)}$. (Note that this step skips at least over the literal $B^{\mathrm{sfac}(C)}$).*

(3) *Next apply Backtrack to S_3, i.e., $S_3 \Rightarrow^{\mathit{Backtrack}}_{\mathit{SCL\text{-}SUP}} S_4$, where $S_4 = (\Gamma_0; N^0; U \cup \{E_2\}; \beta; k_1 - 1; \top)_{(j,C,\gamma)}$.*

(4a) *If L_1 is a negative literal, continue with the following rule applications. Let D be the smallest clause in $N^0 \cup U$ with maximum literal* comp$(L_1) =$ *B_1 and $\Gamma_0 \not\models D$. Then Propagate B_1 from D, and apply Conflict to E_2, i.e., $S_4 \Rightarrow^{\mathit{Propagate}}_{\mathit{SCL\text{-}SUP}} S_4'' \Rightarrow^{*\,\mathit{Conflict}}_{\mathit{SCL\text{-}SUP}} S_5$, where $S_5 = (\Gamma_0, B_1^{\mathrm{sfac}(D)}; N^0; U \cup \{E_2\}; \beta; k_1 - 1; E_2)_{(j,D,\gamma)}$.*

(4b) *If L_1 is a positive literal (i.e., $L_1 = B$) and Conflict is not applicable to $S_5 = (\Gamma_0, B_1^{k_1}; N^0; U \cup \{E_2\}; \beta; k_1; \top)_{(j_2,E_2,\gamma')}$, then decide B, i.e., $S_4 \Rightarrow^{\mathit{Decide}}_{\mathit{SCL\text{-}SUP}} S_5$, where $j_1 + 1$ is the number of occurrences of B_1 in E_2, $j_2 = j + j_1$, and γ' is the same as γ except that $\gamma'(E_2) = \mathrm{sfac}(E_2)$.*

(4c) *If L_1 is a positive literal (i.e., $L_1 = B$) and E_3 is the smallest clause in $N^0 \cup U$ that is false in $S_5' = (\Gamma_0, B_1^{\mathrm{sfac}(E_2)}; N^0; U; \beta; k_1 - 1; \top)_{(j,E_2)}$, then propagate B_1 and apply Conflict to E_3, i.e., $S_4 \Rightarrow^{\mathit{Propagate}}_{\mathit{SCL\text{-}SUP}} S_5' \Rightarrow^{\mathit{Conflict}}_{\mathit{SCL\text{-}SUP}} S_5$, where $S_5 = (\Gamma_0, B_1^{\mathrm{sfac}(E_2)}; N^0; U; \beta; k_1 - 1; E_3)_{(j_2,E_2,\gamma')}$, $j_1 + 1$ is the number of occurrences of B_1 in E_2, $j_2 = j + j_1$, and γ' is the same as γ except that $\gamma'(E_2) = \mathrm{sfac}(E_2)$.*

A (potentially empty) sequence of SCL rule applications according to SCL-SUP is called an atomic sequence *of SCL-SUP steps if it starts from a state S_0 and ends in a state S_5 outlined in the cases (2a) and (5a-c).*

The second part of the strategy simulates the actual inferences resulting from a minimal false clause found in step (2b) of Definition 8 or found in steps (4a) and (4c) of Definition 10. These inferences always correspond to Superposition Left steps of the SUP-MO strategy that resolve minimal false clauses E' in N^i with maximal literal $\neg B$ with the clause C' in N^i that produced B. Note however that SCL-SUP may combine several Superposition Left steps of the SUP-MO strategy into one new learned clause. This is the case whenever the maximal literal $\neg B$ in the minimal false clause E' in N^i is not strictly maximal. In this case, the next minimal false clause E'' will always correspond to the last inferred clause, the maximal literal of this clause will still be $\neg B$, the clause producing B will be again C', and therefore the next Superposition Left partner of E'' is also again C'. Moreover, all of the skipped inferences are actually redundant with respect to the final inference E_2' in this chain, which explains why SCL-SUP is still capable of simulating SUP-MO although it skips the intermediate inferences. The actual

SCL-SUP clause E_2 corresponding to final SUP-MO inference E'_2 is computed in the steps (1) and (2) of Definition 10 with greedy applications of the rules Resolve and Factorize. The following steps of Definition 10 take care of the four different cases how E'_2 changes the model and minimal false clause in N^j. The first case is that $E'_2 = \bot$ so SUP-MO has reached a final state. This case is handled by step (2a) of Definition 10 that simply empties the trail with applications of the rule Skip so the resulting SCL state has the form of a SCL-SUP final state. The second case is that the maximal literal L_1 in E'_2 is negative. In this case, the model for N^i and N^j is still the same and just the minimal false clause changes to E'_2. This case is handled by steps (2b)–(4a) of Definition 10 that Backtrack before $\text{comp}(L_1)$ was decided, propagate it instead and apply Conflict to E_2. In the third and fourth case the maximal literal L_1 in E'_2 is positive. In this case, the model for N^i and N^j actually changes because E'_2 is always productive. Case (2b)–(4b) of Definition 10 handles the case where producing L_1 leads to no new minimal false clause, and case (2b)–(4c) of Definition 10 handles the case where it does. Both cases work symmetrically to steps (2a) and (2b) of Definition 8.

Example 11. We continue Example 9 to demonstrate cases (1)→(4a) and (1)→(2a) of the second part of the SCL-SUP strategy. We left the runs in the SCL state $([P(a)^1, \neg P(b)^2, Q(a)^{\text{sfac}(C_3)}], N^0, \emptyset, \beta, 2, C_3)_{(1,C_2,\gamma_1)}$ that simulates the superposition state N^1, where

$$\begin{aligned} N^1 = \{&(C_1) P(a), \quad (C_2)\ \neg P(b) \vee Q(a), \quad (C_3)\ \neg P(a) \vee Q(a) \vee Q(a), \\ &(C_4)\ P(a) \vee \neg Q(a), \quad (C_5)\ \neg P(a) \vee \neg Q(a), \quad (C_6)\ \neg P(a) \vee Q(a)\} \end{aligned}$$

and C_5 became the minimal false clause in N^1 after C_1 and C_6 produced together the partial model $\{P(a), Q(a)\}$. SUP-MO continues from the state N^1 by applying Superposition Left to C_5 and C_6. In the new state $N^2 = N^1 \cup \{(C_7)\ \neg P(a) \vee \neg P(a)\}$ the new clause order is $C_1 \prec C_7 \prec C_6 \prec C_2 \prec C_3 \prec C_4 \prec C_5$ and after constructing the model for C_1, which produces again $P(a)$, the clause C_7 becomes again the minimal false clause. SCL-SUP follows (1) of Definition 10 and applies Resolve to C_5 and $\text{sfac}(C_3) = C_6$, resulting in the state $([P(a)^1, \neg P(b)^2, Q(a)^{\text{sfac}(C_3)}], N^0, \emptyset, \beta, 2, C_7)_{(2,C_2,\gamma_1)}$. Then SCL-SUP continues with steps (2b) and (3) by applying Skip twice and Backtrack once to jump to the state $(\epsilon, N^0, \{C_7\}, \beta, 0, \top)_{(2,C_2,\gamma_1)}$. Next, SCL-SUP continues with step (4a) because the maximal literal of C_7 is $\neg P(a)$ and therefore negative. This means SCL-SUP will add $P(a)$ again to the trail but this time by applying Propagate to C_1 and afterwards it applies Conflict to C_7. The resulting state $([P(a)^{\text{sfac}(C_1)}], N^0, \{C_7\}, \beta, 0, C_7)_{(2,C_1,\gamma_1)}$ matches again the SUP-MO state N^2.

SUP-MO continues from the state N^2 by applying Superposition Left to C_7 and C_1, resulting in $N^3 = N^2 \cup \{(C_8) \neg P(a)\}$. Since C_8 has the same maximal literal as C_7 it becomes automatically the next minimal false clause in N^3. As a result, SUP-MO applies Superposition Left to C_8 and C_1, which returns $N^5 = N^3 \cup \{(C_9)\ \bot\}$ a final state that proves the unsatisfiability of N^0. Meanwhile, SCL-SUP simulates both Superposition Left steps with one atomic SCL-SUP sequence. It starts with step (1) of Definition 10 and applies Resolve twice, resulting in the state $([P(a)^1, \neg P(b)^2, Q(a)^{\text{sfac}(C_3)}], N^0, \emptyset, \beta, 2, \bot)_{(4,C_2,\gamma_1)}$. Then it continues with step (2a) of Definition 10 and applies Skip until the trail is empty. The resulting state $(\epsilon, N^0, \emptyset, \beta, 2, \bot)_{(4,\bot,\gamma_1)}$ is a final state and proves unsatisfiability of N^0.

Example 12. The next example demonstrates the atomic sequence (1)→(4b) of the second part of the SCL-SUP strategy. Let N^0 be our initial set of clauses:

$$N^0 = \{(C_1)P(a), \quad (C_2)\ \neg P(b), \quad (C_3)\ \neg P(a) \vee Q(a), \quad (C_4)\ P(b) \vee \neg Q(a)$$

As superposition ordering, we choose an LPO with precedence $a \prec b \prec P \prec Q$. This means that the atoms are ordered $P(a) \prec P(b) \prec Q(a) \prec Q(b)$ and the clauses in N^0 are ordered $C_1 \prec C_2 \prec C_3 \prec C_4$. In order to keep the example short, we skip the initial SCL-SUP steps and continue directly with the state $S = ([P(a)^1, \neg P(b)^2, Q(a)^{\mathrm{sfac}(C_3)}], N^0, \emptyset, \beta, 2, C_4)_{(0,C_3,\gamma)}$, where $\gamma(C) = C$ for all clauses C and $\beta = Q(b)$. This state simulates the superposition state N^0 upto the model construction for C_3, where $N^0_{C_3} \cup \delta_{C_3} = \delta_{C_1} \cup \delta_{C_3} = \{P(a), Q(a)\}$ and C_4 is the minimal false clause. SUP-MO continues from the state N^0 by applying Superposition Left to C_4 and C_3. In the new state $N^1 = N^0 \cup \{(C_5)\ \neg P(a) \vee P(b)\}$ the new clause order is $C_1 \prec C_5 \prec C_2 \prec C_3 \prec C_4$ and the partial model upto C_5 is $N^0_{C_5} \cup \delta_{C_5} = \delta_{C_1} \cup \delta_{C_5} = \{P(a)\} \cup \{P(b)\}$, which turns C_2 into the next minimal false clause. SCL-SUP simulates the above steps by following the atomic sequence (1)→(4b) of Definition 10. The result is the state $([P(a)^1, P(b)^{\mathrm{sfac}(C_5)}], N^0, \{C_5\}, \beta, 1, C_2)_{(1,C_5,\gamma)}$ matching again our current superposition state and model.

Without clause C_2, SCL-SUP would apply the atomic sequence (1)→(4a) of Definition 10 to S, resulting in the state $([P(a)^1, P(b)^2], N^0 \setminus \{C_2\}, \{C_5\}, \beta, 2, \top)_{(1,C_5,\gamma)}$. This matches the state $N^1 \setminus \{C_2\}$ and its partial model upto C_5 that is still the same as for N^1 with the exception that it does not lead to a minimal false clause.

In order to actually show that every SCL-SUP run simulates a SUP-MO run, we need to prove three properties. The first property is that each state visited by an SCL-SUP run must simulate a state visited by the corresponding SUP-MO run. Note that this property does not yet say anything about the order in which SCL-SUP simulates the SUP-MO states. This property can also be seen as a soundness argument for our strategy.

Lemma 13 (Initial SCL State Simulates Initial Superposition State). *The initial SCL state $(\epsilon; N^0; \emptyset; \beta; 0; \top)_{(0,\bot,\gamma)}$ simulates the initial superposition state N^0 and the model construction upto $N^0_\bot \cup \delta_\bot$*

Lemma 14 (SCL-SUP Preserves Simulation). *Let the SCL state $S = (\Gamma; N^0; U; \beta; k; E)_{(i,C,\gamma)}$ simulate the superposition state N^i and the corresponding model construction upto $N^i_{C'} \cup \delta_{C'}$, where $C' = \gamma(C)$. Let the SCL state $S' = (\Gamma'; N^0; U'; \beta; k'; E')_{(j,D,\gamma')}$ be the result of one atomic sequence of SCL-SUP steps. Then there exists a clause $D' \in N^j$ with $\gamma'(D) = D'$ and S' simulates the superposition state N^j and the model construction upto $N^j_{D'} \cup \delta_{D'}$.*

The second property is that each atomic sequence of SCL-SUP steps always makes progress in the simulation. This means that each atomic sequence of SCL-SUP steps either advances the superposition state N^i simulated by the current SCL state $S = (\Gamma; N^0; U; \beta; k; E)_{(i,D,\gamma)}$, i.e., it increases the annotated i, or it still simulates the same superposition state N^i but advances the simulation of the model construction operator, i.e. it increases the annotated clause C and keeps i the same. Note that it can actually happen that an atomic sequence of SCL-SUP steps skips over several superposition states. This property can also be seen as a termination argument for our strategy because SUP-MO always terminates on ground clause sets.

Lemma 15 (SCL-SUP Advances the Simulation). *Let the SCL state $S = (\Gamma; N^0; U; \beta; k; E)_{(i,D,\gamma)}$ simulate the superposition state N^i and the model construction upto $N^i_D \cup \delta_D$. Let the SCL state $S' = (\Gamma'; N^0; U'; \beta; k'; E')_{(j,D',\gamma')}$ be the next state reachable by one atomic sequence of SCL-SUP steps. Then either $i < j$ or $i = j$ and $\gamma' = \gamma$ and $D \prec_\gamma D'$.*

The last missing property shows that the SCL-SUP strategy can always advance the current SCL state whenever the simulated superposition state can be advanced by the SUP-MO strategy. This means SCL-SUP is never stuck when SUP-MO can still progress. These properties hold because the simulation invariants in Definition 7 either correspond to a correct final state or they satisfy the preconditions of Definition 8 or Definition 10. This property can also be seen as a partial correctness argument for our strategy.

Lemma 16 (SCL-SUP Correctness of Final States). *Let the SCL state $S = (\Gamma; N^0; U; \beta; k; E)_{(i,D,\gamma)}$ simulate the superposition state N^i and the model construction upto $N^i_{\gamma(D)} \cup \delta_{\gamma(D)}$. Let there be no more states reachable from S following an atomic sequence of SCL-SUP steps. Then S is a* final state*, i.e., either (i) $E = \bot$, $D = \bot$, $\bot \in N^i$, and N^0 is unsatisfiable or (ii) $\Gamma \models N^0$.*

We can also show that any SCL-SUP run is also a regular run. Although this is not strictly necessary for the simulation proof, it is beneficial because it means that SCL-SUP inherits many properties that hold for SCL restricted to a regular strategy. For instance, that all learned clauses are non-redundant and that SCL-SUP always terminates.

Lemma 17 (SCL-SUP is a Regular SCL Strategy). *SCL-SUP is a regular SCL strategy if it is executed on a state $S = (\Gamma; N^0; U; \beta; k; E)_{(i,C,\gamma)}$ that simulates a superposition state N^i and the corresponding model construction upto $N^i_{\gamma(C)} \cup \delta_{\gamma(C)}$.*

4 Conclusion

We have shown that the SCL(FOL) calculus [9] can simulate model driven superposition [1] refutations deriving only non-redundant clauses. The superposition calculus cannot simulate SCL refutations due to its static a priori ordering.

In general, an SCL(FOL) learned clause is generated out of several resolution and factorization steps. From this perspective the SCL(FOL) calculus is more general and flexible than the superposition calculus. Furthermore, it only generates non-redundant clauses whereas any superposition implementation generates redundant clauses due to the syntactic application of the superposition inference rules.

Selection in superposition can also be simulated, but requires an additional branch in the SCL-SUP strategy, because selection of non-maximal, negative literals by superposition requires a different trail ordering for SCL in order to simulate a respective superposition left inference.

For future work, we plan to lift our simulation result from the ground case to the non-ground case. This lifting will require the extension of the SCL calculus by an additional rule that learns clauses that are computed as intermediate steps during the conflict analysis. This rule was left out of previous versions of SCL because we would never use it in a CDCL inspired SCL-run and because it would have complicated the termination and non-redundancy proofs for SCL. Nevertheless, we are confident that the rule can be designed in such a way that all properties of the original calculus still hold.

Considering the extension to the non-ground case, this result can be used in various directions. It can be used to develop an alternative implementation of the superposition calculus. Given a fixed ordering, the trail can be developed according to the ordering, generating only non-redundant superposition inferences. On the other hand, the concept of finite saturation can be kept this way preserving a strong mechanism for detecting satisfiability. Secondly, the result means that SCL can be used to naturally combine propagation driven reasoning with fixed ordering driven reasoning. This might overcome some of the issues of the current first-order portfolio approaches implemented in the state-of-the-art provers.

Another calculus contained in first-order reasoning portfolios is InstGen [12, 15]. It abstracts a first-order clause set to propositional logic via a grounding with a single constant. In case a CDCL sat solver proves the abstraction unsatisfiable, the first-order clause set is unsatisfiable too. For otherwise, the model found on the propositional level triggers an instantiation inference of a first-order clause. The instance rules out the before found propositional model modulo the abstraction.

The CDCL model building after grounding can be simulated via a respective SCL trail. This will then lead to a stuck state if SCL is restricted to the InstGen grounding. Now let C be the false first-order clause selected by InstGen for an instance. Then the SCL stuck state can be extended to a conflict state for C. Then SCL will not learn an instance of C, but a related clause that also rules out the previously found model on the propositional level. This way the relationship between InstGen and SCL can be investigated as well.

Acknowledgements. We thank our reviewers for their careful reading and constructive comments.

References

1. Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. J. Logic Comput. **4**(3), 217–247 (1994). Revised version of Max-Planck-Institut für Informatik technical report, MPI-I-91-208, 1991
2. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, chap. 2, pp. 19–99. Elsevier, Amsterdam (2001)
3. Bromberger, M.: A sorted datalog hammer for supervisor verification conditions modulo simple linear arithmetic. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13243, pp. 480–501. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_27
4. Bromberger, M., Fiori, A., Weidenbach, C.: Deciding the Bernays-Schoenfinkel fragment over bounded difference constraints by simple clause learning over theories. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) VMCAI 2021. LNCS, vol. 12597, pp. 511–533. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67067-2_23
5. Bromberger, M., Gehl, T., Leutgeb, L., Weidenbach, C.: A two-watched literal scheme for first-order logic. In: Konev, B., Schon, C., Steen, A. (eds.) Proceedings of the Workshop on Practical Aspects of Automated Reasoning Co-located with the 11th International Joint Conference on Automated Reasoning (FLoC/IJCAR 2022), CEUR Workshop Proceedings, Haifa, Israel, 11–12 August 2022, vol. 3201. CEUR-WS.org (2022)
6. Bromberger, M., Jain, C., Weidenbach, C.: SCL(FOL) can simulate non-redundant superposition clause learning (2023)
7. Bromberger, M., Leutgeb, L., Weidenbach, C.: An efficient subsumption test pipeline for bs(lra) clauses. In: Blanchette, J., Kovacs, L., Pattinson, D. (eds.) Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Held as Part of the Federated Logic Conference, Proceedings. LNCS, vol. 13385, pp. 147–168. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-10769-6_10
8. Bromberger, M., Schwarz, S., Weidenbach, C.: Exploring partial models with SCL. In: Konev, B., Schon, C., Steen, A. (eds.) Proceedings of the Workshop on Practical Aspects of Automated Reasoning Co-located with the 11th International Joint Conference on Automated Reasoning (FLoC/IJCAR 2022), CEUR Workshop Proceedings, Haifa, Israel, 11–12 August 2022, vol. 3201 (2022)
9. Bromberger, M., Schwarz, S., Weidenbach, C.: SCL(FOL) revisited (2023). https://doi.org/10.48550/ARXIV.2302.05954. https://arxiv.org/abs/2302.05954
10. Desharnais, M.: A formalization of the SCL(FOL) calculus: Simple clause learning for first-order logic. Archive of Formal Proofs (2023). https://isa-afp.org/entries/Simple_Clause_Learning.html, Formal proof development
11. Fiori, A., Weidenbach, C.: SCL clause learning from simple models. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 233–249. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_14
12. Ganzinger, H., Korovin, K.: New directions in instatiation-based theorem proving. In: Abramsky, S. (ed.) 18th Annual IEEE Symposium on Logic in Computer Science, LICS 2003, pp. 55–64. IEEE Computer Society (2003)
13. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Leech, I. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press, Oxford (1970)

14. Korovin, K.: iProver – an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_24
15. Korovin, K.: Inst-Gen – a modular approach to instantiation-based automated reasoning. In: Voronkov, A., Weidenbach, C. (eds.) Programming Logics. LNCS, vol. 7797, pp. 239–270. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37651-1_10
16. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
17. Leidinger, H., Weidenbach, C.: SCL(EQ): SCL for first-order logic with equality. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, 8–10 August 2022, Proceedings. Lecture Notes in Computer Science, vol. 13385, pp. 228–247. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-10769-6_14
18. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, chap. 7, pp. 371–443. Elsevier (2001)
19. Schlichtkrull, A., Blanchette, J.C., Traytel, D., Waldmann, U.: Formalization of bachmair and ganzinger's ordered resolution prover. Archive of Formal Proofs (2018). https://isa-afp.org/entries/Ordered_Resolution_Prover.html, Formal proof development
20. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 495–507. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_29
21. Waldmann, U., Tourret, S., Robillard, S., Blanchette, J.: A comprehensive framework for saturation theorem proving. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, 1–4 July 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12166, pp. 316–334. Springer, Heidelberg (2020). https://doi.org/10.1007/978-3-030-51074-9_18
22. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 140–145. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_10

Formal Reasoning About Influence in Natural Sciences Experiments

Florian Bruse, Martin Lange(✉), and Sören Möller

Theoretical Computer Science/Formal Methods, University of Kassel, Kassel, Germany
martin.lange@uni-kassel.de

Abstract. We present a simple calculus for deriving statements about the local behaviour of partial, continuous functions over the reals, within a collection of such functions associated with the elements of a finite partial order. We show that the calculus is sound in general and complete for particular partial orders and statements. The motivation for this work is drawn from an attempt to foster digitalisation in secondary-eduction classrooms, in particular in experimental lessons in natural science classes. This provides a way to formally model experiments and to automatically derive the truth of hypotheses made about certain phenomena in such experiments.

Keywords: formal modelling · proof system · continuous functions · completeness

1 Introduction

Formal reasoning using proof rules is a well-established mechanism for explaining and deriving the truth of statements, both in general-purpose first- and higher-order logics [2,16] as well as special-purpose logics in arithmetic [5], knowledge discovery [15], program verification [13] etc. Here we are concerned with the problem of proving statements about the local "behaviour" of certain real-valued functions. A proof calculus for such simple statements may be interesting purely for its logical (meta-)properties. There is, however, also a very concrete motivation for this work: digitalisation of experiments in natural sciences in secondary-education classrooms. Studies show how digitalisation can benefit such teaching-learning environments [10,18], not least by channeling pupils' interaction through a software tool to enforce better learning [11].

In classes of natural sciences like biology, physics and chemistry, pupils are often taught some background knowledge about particular subjects which they then need to put to the test experimentally. For this, they are given a *research question* which typically asks them to discover and formulate a particular phenomenon in form of a so-called *hypothesis*, and to validate its correctness experimentally. Take for instance as an "experiment" in a physics class the standard European alternating current at 230 V 50 Hz. The way that voltage fluctuates

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 153–169, 2023.
https://doi.org/10.1007/978-3-031-38499-8_9

over time – in other words: time *influences* voltage – and voltage induces (resp. *influences*) a current, forms the background theory, and a research question could for instance be: *how does the current change over time*? We aim to provide digital technology that can answer such questions automatically in order to give valid feedback to a pupil about their success in this task.

Formal models for processes from natural sciences have been proposed in the literature [19], like Petri nets [6,12] or hybrid automata [1,3]. They allow for precise modelling of experiments; the price to pay is that of undecidability of model checking already, let alone validity checking. Moreover, they rely on exact knowledge about the nature of influences in such experiments, and this can often only be described by differential equations. Hence, determining correctness of a hypothesis requires sophisticated algebraic or numerical methods.

Here, we model experiments abstractly as *influence schemes*, that is sets $\mathcal{C}$ of statements about certain parts of an influence, allowing them to be built from observations for instance. Correctness of a hypothesis H then is the question of whether H logically follows from $\mathcal{C}$. We provide the framework for modelling experiments and hypotheses about influences in the form of a simple language of statements, a formal semantics via collections of partial continuous functions, and a proof calculus for logical consequence in this language. We show that it is sound in general, complete for a large and useful class of hypotheses and experiment models, i.e. influence schemes, and that it is polynomial-time decidable.

The completeness proof uses elements that are similar to constructions for general logics. A key ingredient is normalisation, essentially a saturation process comparable to the construction of Hintikka or maximally consistent sets, cf. [7, 17]. Another one is the effective construction of countermodels for such saturated sets, cf. [8,9,14]. The details of these constructions are of course tailored to the specifics of the mixed discrete-continuous structures here, dealing with properties of collections of (partial) continuous functions associated with pairs of elements of some underlying finite partial order.

The paper is organised as follows. Section 2 introduces the mathematical basics in terms of functions on the reals, statements, influence schemes, hypotheses etc. Section 3 presents the proof calculus including its soundness. Section 4 begins by showing that the proof calculus is generally incomplete, as the relatively simple statements cannot make assertions capturing certain phenomena arising with functions on the reals. We then develop a restriction on influence schemes and show that completeness does hold in this case. The full proofs of technical lemmas are omitted for reasons of space restriction. Section 5 discusses the computational problem of proof search. Section 6 concludes with remarks on further work.

2 Modelling Influence

Statements and Influence Schemes. In all of the following, $\mathcal{V} = \{a, b, \ldots\}$ denotes a finite set of *variables*, and we assume that these are partially ordered by $\leq$ with $<$ denoting its strict subset.

An *interval* (of reals) is denoted $[x, y]$ for $x, y \in \mathbb{Q} \cup \{-\infty, \infty\}$ with $x \leq y$. Abusing standard notation, we write, e.g. $[-\infty, 10]$ rather than $(-\infty, 10]$ for the set of all real numbers z with $z \leq 10$, since we only consider intervals that are closed at rational bounds (for purposes of effective representation) and semi-open only at infinities. This provides a common notation for intervals and saves us making case distinctions everywhere, depending on the interval bounds.

A $\mathcal{V}$-*statement* is a 5-tuple $S = (a, I, q, I', b)$, typically written as $a \xhookrightarrow{I\,q\,I'} b$, s.t. $a, b \in \mathcal{V}$ with $a < b$, and I, I' are intervals in the sense above. I is called the *domain*, denoted $dom(S)$, and required to be a non-singleton interval. I' is the *range*, denoted $rng(S)$. Finally, $q \in \mathcal{Q} := \{\nearrow, \searrow, \rightarrow, \rightsquigarrow\}$ is called a *behaviour*. It describes a gradient of the influence abstractly as either *monotonic*, *antitonic*, *constant* or *arbitrary*. When the variables a, b involved in the statement S are clear from or irrelevant for the context, we also often simply write $\xhookrightarrow{I\,q\,I'}$.

The statement S is used to formalise the assertion "*variable $a$influences variable b on the interval I in a way described by q, s.t. varying the value for a in this interval results in b taking values from the interval I'.*"

A $\mathcal{V}$-*influence scheme*, or simply influence scheme if $\mathcal{V}$ is clear from the context, is a finite set $\mathcal{C}$ of $\mathcal{V}$-statements. Intuitively, an influence scheme describes the way that certain variables influence each other in an abstract way.

Example 1. We build an influence scheme for the AV-voltage experiment. The relevant variables are $\mathtt{t}$ for *time*, $\mathtt{v}$ for *voltage* and $\mathtt{c}$ for *current*, ordered by $\mathtt{t} < \mathtt{v} < \mathtt{c}$. A theory of how voltages alternates over time (in the standard European alternating 230 V/50 Hz setting) and how it induces a current at a resistance of 326 Ω can be formalised as follows. Remember that a scheme is a finite set of statements like $\mathtt{t} \xhookrightarrow{[0,5]\ \nearrow\ [0,326]} \mathtt{v}$ etc. Each can easily be visualised as a rectangle in the 2-dimensional plane for the pair of involved variables: horizontal and vertical edges determine domain and range, and the behaviour can be shown as a label on the rectangle. A particular influence scheme $\mathcal{C}$ with 20 statements is shown in Fig. 1 as grey rectangles in this way. The behaviours in the graph in the middle are left out for better visibility; they are all supposed to be $\nearrow$.

The orange lines in the graphs of Fig. 1 represent a so-called *influence experiment*, as it will be explained below. At this point, it can be used to show that influence schemes as formal models of experiments can be obtained through data sampling. Note how the borders of the rectangles in the scheme $\mathcal{C}$ coincide with values of the functions represented by the orange lines in most cases.

Note that the scheme $\mathcal{C}$ shown in Fig. 1 contains no statements for the pair $(\mathtt{t}, \mathtt{c})$ of variables. This does not mean that time does not influence current in this scheme: clearly, if time influences voltage, and voltage influences current, then time executes some influence on current. Hence, a valid question asks whether the statement H shown as a blue rectangle follows logically from the scheme $\mathcal{C}$ in the sense that whenever time influences voltage and voltage influences current in the way described by $\mathcal{C}$, does time then also influence current in the way described by H? We use the letter H for such a statement as it plays the role of a *hypothesis*: in logical terms it is just a statement, but from an application point of view it is special in that it signifies an implicit question after its truth with respect to a scheme.

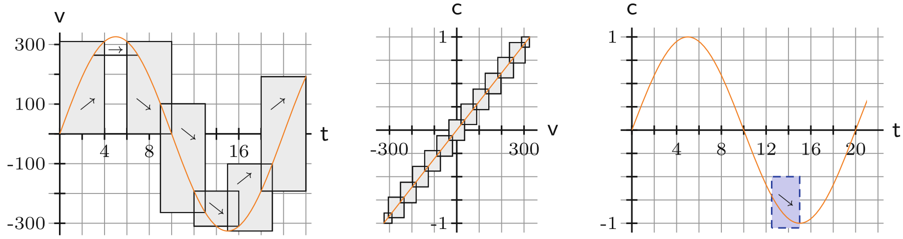


Fig. 1. An influence scheme (grey rectangles), a hypothesis (blue dashed rectangle) and an influence experiment (orange lines) between time, voltage and current. (Color figure online)

A Formal Semantics. In order to give a well-defined meaning to the question whether H follows from $\mathcal{C}$ for a given scheme $\mathcal{C}$ and hypothesis H, we introduce a formal interpretation of statements in so-called *influence experiments*. We need to recall and define a few technicalities about functions over the reals.

An *influence* is a function $f : \mathbb{R} \rightharpoonup \mathbb{R}$ s.t. $dom(f)$ is a non-singleton interval in the sense above, and f is continuous on its domain in the usual sense. We write $f(x) = \bot$ if $x \notin dom(f)$. When composing partial functions we assume undefined values to be absorbing, i.e. $g(f(x)) = \bot$ if $f(x) = \bot$.

An influence f is called *monotonic*, *antitonic* or *constant* on $[x, y] \subseteq dom(f)$, if for all $z, z' \in [x, y]$ with $z \leq z'$ we have $f(z) \leq f(z')$, respectively $f(z) \geq f(z')$ and $f(z) = f(z')$. It *satisfies* the statement $S = \xhookrightarrow{[x,y]\,q\,[x',y']}$, written $f \models S$, if the following two conditions are met.

1. $f(z) \in [x', y']$ for all $z \in [x, y]$.
2. $q = \nearrow$ and f is monotonic on $[x, y]$, or $q = \searrow$ and f is antitonic on $[x, y]$, or $q = \rightarrow$ and f is constant on $[x, y]$, or $q = \rightsquigarrow$.

Since every constant function is monotonic and antitonic, and each of these is also an arbitrary one, we naturally obtain a partial order $\preceq$ on behaviours that features unique infima and suprema, shown in Fig. 3. Note that, whenever $f \models \xhookrightarrow{I\,q\,I'}$ and $q \preceq q'$ then also $f \models \xhookrightarrow{I\,q'\,I'}$.

We are now ready to define the formal semantics of influence schemes.

Definition 1. *Let $\mathcal{V}$ be as above. A $\mathcal{V}$-*influence experiment *is a collection $\mathcal{F}$ of influences, namely one function $\mathcal{F}_{a,b}$ for each pair (a, b) s.t. $a < b$, altogether satisfying the following* coherence property *(CP).*

- *For all $a, b, c \in V$ s.t. $a < b$, $b < c$ and all $x \in \mathbb{R}$: $\mathcal{F}_{a,c}(x) = \mathcal{F}_{b,c}(\mathcal{F}_{a,b}(x))$.*

$\mathcal{F}$ satisfies the $\mathcal{V}$-statement $S = a \xhookrightarrow{I\,q\,I'} b$, written $\mathcal{F} \models S$, if $\mathcal{F}_{a,b} \models \xhookrightarrow{I\,q\,I'}$. $\mathcal{F}$ satisfies the $\mathcal{V}$-influence scheme $\mathcal{C}$, written $\mathcal{F} \models \mathcal{C}$, if $\mathcal{F} \models S$ for all $S \in \mathcal{C}$.

CP together with the absorption of $\bot$ in function composition is the reason for demanding the variables to be partially ordered: $\mathcal{F}_{a,a}$, for any variable a would have to be the total identity function to satisfy CP. And then we would have $\mathcal{F}_{b,a} = \mathcal{F}^{-1}_{a,b}$ for any a, b. Thus, by demanding that $\mathcal{F}_{a,b}$ is only defined whenever $a < b$ we avoid problems arising with non-invertible functions.

Example 2. Figure 1 shows a particular time-voltage-current experiment $\mathcal{F}$ as three influences drawn as orange graphs. It represents the way that voltage alternates in time along a sine curve with amplitude $230 \cdot \sqrt{2} \approx 326$ V and frequency 50 Hz. Electric current depends linearly on voltage in this experiment, with a factor of $\frac{1}{326}$ used here suggesting an electrical resistance of 326 Ω. The coherence property then demands a third influence $\mathcal{F}_{\mathsf{t,c}}$ as their composition on the domain of $\mathcal{F}_{\mathsf{t,v}} = [0, \infty]$ which is also a sine curve.

Let $\mathcal{C}$ be the influence scheme shown in Fig. 1 and introduced in Example 1. Clearly $\mathcal{F} \not\models \mathcal{C}$ because $\mathcal{F}$ does not satisfy the second (degenerate) rectangle representing the statement $\mathsf{t} \xhookrightarrow{[3,7] \,\rightarrow\, [264,264]} \mathsf{v}$ and neither the fifth representing $\mathsf{t} \xhookrightarrow{[12,16] \,\searrow\, [-310,-192]} \mathsf{v}$. This is because $\mathcal{F}_{\mathsf{t,v}}$ is neither constant on $[3, 7]$ nor antitonic on $[12, 16]$, and because it assumes values outside of the statements' ranges on these domains, e.g. $\mathcal{F}_{\mathsf{t,v}}(5) = 326 \notin [264, 264]$ and $\mathcal{F}_{\mathsf{t,v}}(15) = -326 \notin [-310, -192]$.

Note that satisfaction of a statement S by an influence f means that the graph of f enters the rectangle representing S through its left edge and leaves it only through its right edge, and within this rectangle it displays the behaviour stated in S. This is the case for instance for the hypothesis H drawn as a blue rectangle: $\mathcal{F} \models H$ indeed. But this does not allow any conclusion to be drawn about whether H follows from $\mathcal{C}$ in any way.

The interpretation of an influence scheme through influence experiments naturally gives rise to a notion of logical consequence: we say that the $\mathcal{V}$-statement H *follows* from the $\mathcal{V}$-influence scheme $\mathcal{C}$, written $\mathcal{C} \models H$, if $\mathcal{F} \models H$ for all $\mathcal{V}$-influence experiments s.t. $\mathcal{F} \models \mathcal{C}$. Thus, an influence scheme $\mathcal{C}$ can be seen as a finite representation of an (uncountable) number of $\mathcal{V}$-experiments, which yields the abstract nature of these schemes as mentioned in the introduction.

The semantics also gives rise to a natural notion of equivalence between schemes: $\mathcal{C}$ and $\mathcal{C}'$ are *equivalent*, written $\mathcal{C} \equiv \mathcal{C}'$, if for all $\mathcal{F}$ we have $\mathcal{F} \models \mathcal{C}$ iff $\mathcal{F} \models \mathcal{C}'$. Note that this is the case iff for all hypotheses H we have $\mathcal{C} \models H$ iff $\mathcal{C}' \models H$. Equivalent schemes can therefore be seen as (possibly different) descriptions of the same experimental setup, up to a certain amount of imprecision determined by the description of the experimental setup through discrete statements.

3 The Calculus of Influence

The concept of consequence between a scheme and a hypothesis provides the foundations for a logical approach to modelling experimental setups and correctness of hypotheses w.r.t. them. Ideally, the consequence relation $\models$ would be decidable, since this would provide a way to automatically check the correctness of a hypothesis w.r.t. a given scheme. In this section we develop a proof-theoretic characterisation of $\models$ in terms of a provability predicate $\vdash$. Ideally, $\vdash$ would be sound and complete w.r.t. $\models$, i.e. a statement would follow from an influence scheme iff it is provably derivable from it. Then decidability of $\vdash$ (cf. Sect. 5)

$$(\mathsf{F})\ \frac{}{S} \text{ if } S \in \mathcal{C}$$

$$(\mathsf{G})\ \frac{a \xhookrightarrow{[x,y]\; q\; I} b \qquad a \xhookrightarrow{[x',y']\; q'\; I'} b}{a \xhookrightarrow{[y,x'] \;\rightsquigarrow\; [-\infty,\infty]} b} \text{ if } y < x'$$

$$(\mathsf{T})\ \frac{a \xhookrightarrow{I\; q\; I_1} b \qquad b \xhookrightarrow{I_2\; q'\; I'} c}{a \xhookrightarrow{I\; q \otimes q'\; I'} c} \text{ if } I_1 \subseteq I_2$$

$$(\mathsf{I}^-)\ \frac{a \xhookrightarrow{I_1\; q\; I_2} b}{a \xhookrightarrow{I_1'\; q\; I_2'} b} \text{ if } I_1' \subseteq I_1, I_2 \subseteq I_2'$$

$$(\mathsf{L}^+_{\nearrow})\ \frac{a \xhookrightarrow{[x,y]\; \nearrow\; [l,u]} b \qquad a \xhookrightarrow{[y,z]\; q\; [l',u']} b}{a \xhookrightarrow{[x,y]\; \nearrow\; [l,\min(u,u')]} b}$$

$$(\mathsf{I}^+)\ \frac{a \xhookrightarrow{I_1\; q\; I_1'} b \qquad a \xhookrightarrow{I_2\; q'\; I_2'} b}{a \xhookrightarrow{I_1 \cap I_2\;\; \inf_{\preceq}(q,q')\; I_1' \cap I_2'} b}$$

$$(\mathsf{L}^+_{\searrow})\ \frac{a \xhookrightarrow{[x,y]\; \searrow\; [l,u]} b \qquad a \xhookrightarrow{[y,z]\; q\; [l',u']} b}{a \xhookrightarrow{[x,y]\; \searrow\; [\max(l,l'),u]} b}$$

$$(\mathsf{J})\ \frac{a \xhookrightarrow{[x,y]\; q\; I'} b \qquad a \xhookrightarrow{[y,z]\; q'\; I'} b}{a \xhookrightarrow{[x,z]\;\; \sup_{\preceq}(q,q')\; I'} b}$$

$$(\mathsf{R}^+_{\nearrow})\ \frac{a \xhookrightarrow{[x,y]\; q\; [l,u]} b \qquad a \xhookrightarrow{[y,z]\; \nearrow\; [l',u']} b}{a \xhookrightarrow{[y,z]\; \nearrow\; [\max(l,l'),u']} b}$$

$$(\mathsf{Q}^-)\ \frac{a \xhookrightarrow{I\; q\; I'} b}{a \xhookrightarrow{I\; q'\; I'} b} \text{ if } q \preceq q'$$

$$(\mathsf{R}^+_{\searrow})\ \frac{a \xhookrightarrow{[x,y]\; q\; [l,u]} b \qquad a \xhookrightarrow{[y,z]\; \searrow\; [l',u']} b}{a \xhookrightarrow{[y,z]\; \searrow\; [l',\min(u,u')]} b}$$

$$(\mathsf{C})\ \frac{a \xhookrightarrow{I\; q\; [y,y]} b}{a \xhookrightarrow{I\; \rightarrow\; [y,y]} b}$$

Fig. 2. Proof rules for correctness of a statement w.r.t. an influence scheme $\mathcal{C}$. See Fig. 3 for the definitions of $\preceq$ and $\otimes$.

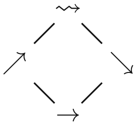
would yield the basis for automatic reasoning about influence in experimental setups.

Henceforth, let $\mathcal{V}$ and a $\mathcal{V}$-influence scheme $\mathcal{C}$ be fixed. We say that a $\mathcal{V}$-statement H is *provable* w.r.t. $\mathcal{C}$, written $\mathcal{C} \vdash H$, if there is a finite proof for H in the proof system whose rules are shown in Fig. 2.

We will briefly explain the intuition behind each of them. The rule (F), which serves as an axiom, essentially states that any statement which is part of the scheme, follows from it. (G) expresses the fact that experiments are comprised of potentially partial functions whose domain is always some interval. It states that any function $\mathcal{F}_{a,b}$ which shows some certain behaviour on the interval $[x, y]$, and some certain behaviour on the interval $[x', y']$ where $y < x'$, must also be defined on the interval $[y', x]$. However, we cannot determine better bounds than infinities on its values, nor a non-arbitrary behaviour there.

Rule (T) expresses the transitivity principle laid out in the coherence property of $\mathcal{V}$-experiments: when a influences b s.t. a-values in I lead to b-values in I_1, and $I_1 \subseteq I_2$, and b-values in I_2 lead to c-values in I', then a-values in I lead to c-values in I'. Moreover, the behaviour of the influence from a to c can be derived from the ones from a to b and from b to c via the multiplication table for $\otimes$ shown in Fig. 3.

Rule (I^-) expresses weakening of statements w.r.t. the involved intervals. Any function which maps values from I_1 to values in I_2 must also do so for values from a subset of I_1, and their range is naturally limited by any superset of I_2. On the other hand, (I^+) represents an important strengthening principle:



	$\nearrow$	$\searrow$	$\rightarrow$	$\rightsquigarrow$
$\nearrow$	$\nearrow$	$\searrow$	$\rightarrow$	$\rightsquigarrow$
$\searrow$	$\searrow$	$\nearrow$	$\rightarrow$	$\rightsquigarrow$
$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightsquigarrow$
$\rightsquigarrow$	$\rightsquigarrow$	$\rightsquigarrow$	$\rightsquigarrow$	$\rightsquigarrow$

Fig. 3. Order $\preceq$ (left) and multiplication $\otimes$ (right) on behaviours.

any function that maps values from I_1 to I'_1 and values from I_2 to I'_2 must map values from $I_1 \cap I'_1$ to $I_2 \cap I'_2$. Note that the rule is only (meaningfully) applicable if $I_1 \cap I'_1 \neq \emptyset$. Moreover, the behaviour on the intersection can be determined from those on the two involved intervals. For instance, if $\mathcal{F}_{a,b}$ is monotonic on I_1 and antitonic on I'_1 then it must be both monotonic and antitonic on $I_1 \cap I'_1$, hence, it must in fact be constant there.

Rules $(\mathsf{L}^+_\nearrow)$–$(\mathsf{R}^+_\searrow)$ express further strengthening principles which are applicable in situations where two statements are made about the behaviour of a function on adjacent intervals. Suppose for instance, that $\mathcal{F}_{a,b}$ maps values from $[x, y]$ monotonically into $[l, u]$, and values from $[y, z]$ somehow into $[l', u']$. In particular, we have $\mathcal{F}_{a,b}(y) \leq u$ since $y \in [x, y]$, and $\mathcal{F}_{a,b}(y) \leq u'$ since $y \in [y, z]$, i.e. $\mathcal{F}_{a,b}(y) \leq \min(u, u')$. By monotonicity, for all z' with $x \leq z' \leq y$ we must have $\mathcal{F}_{a,b}(z') \leq \min(u, u')$ as well. Hence, from the knowledge about the monotonic behaviour of $\mathcal{F}_{a,b}$ on $[x, y]$ and the upper bound on an adjacent interval to the right of it, we can possibly infer a tighter upper bound on the values of $\mathcal{F}_{a,b}$ on $[x, y]$. This is what rule $(\mathsf{L}^+_\nearrow)$ does. The other three rules $(\mathsf{L}^+_\searrow)$, $(\mathsf{R}^+_\nearrow)$ and $(\mathsf{R}^+_\searrow)$ cover the analogous cases of the behaviour being antitonic or the adjacent statement being on the other side.

Rule (J) can be used to infer statements about the behaviour of a function on parts of its domain which are comprised of several intervals. If $\mathcal{F}_{a,b}$ maps values from $[x, y]$ into I_1 with behaviour q, and values from $[y, z]$ into I_2 with behaviour q', then it maps values from $[x, z]$ into $I_1 \cup I_2$, provided that this is an interval. Moreover, the behaviour on the larger interval can be determined from q and q' by simply taking the supremum w.r.t $\preceq$. This is obviously associative, which allows us to write $\sup_{\preceq}(q_1, \ldots, q_n)$ without ambiguity.

Note that (J) is also a weakening rule: for instance, from $S_1 = a \xhookrightarrow{[0,1]\,\rightsquigarrow\,[0,1]} b$ and $S_2 = a \xhookrightarrow{[1,2]\,\rightsquigarrow\,[1,2]} b$ we can infer $S = a \xhookrightarrow{[0,2]\,\rightsquigarrow\,[0,2]} b$, describing any influence $\mathcal{F}_{a,b}$ that maps values from $[0, 2]$ to $[0, 2]$, for instance $\mathcal{F}_{a,b}(x) = 2 - x$. I.e. we have $\mathcal{F} \models S$, but $\mathcal{F} \not\models S_1$ and $\mathcal{F} \not\models S_2$. Likewise, (Q^-) allows the weakening of behaviours. It states that a function which possesses a certain behaviour on an interval also possesses any weaker behaviour on this interval.

At last, rule (C) expresses a simple principle: an influence of variable a onto b whose values can be bounded by a singleton interval, is of constant behaviour.

Example 3. A proof of $\mathcal{C} \vdash H$ for the scheme $\mathcal{C}$ and the hypothesis $H = \mathsf{t} \xhookrightarrow{[12.5,15]\,\searrow\,[-1.05,-0.5]} \mathsf{c}$ shown in Fig. 1 (cf. Example 1) is given in Fig. 4. The

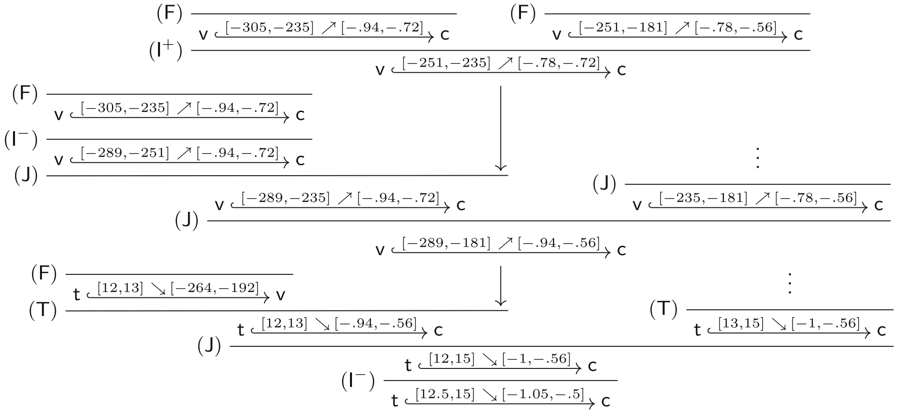

(F)
v [−305,−235] ↗ [−.94,−.72] c
(I+)
(F)
v [−251,−181] ↗ [−.78,−.56] c
v [−251,−235] ↗ [−.78,−.72] c
(F)
v [−305,−235] ↗ [−.94,−.72] c
(I−)
v [−289,−251] ↗ [−.94,−.72] c
(J)
v [−289,−235] ↗ [−.94,−.72] c
(J)
(J)
v [−235,−181] ↗ [−.78,−.56] c
v [−289,−181] ↗ [−.94,−.56] c
(F)
t [12,13] ↘ [−264,−192] v
(T)
t [12,13] ↘ [−.94,−.56] c
(J)
(T)
t [13,15] ↘ [−1,−.56] c
t [12,15] ↘ [−1,−.56] c
(I−)
t [12.5,15] ↘ [−1.05,−.5] c

Fig. 4. Proof of the hypothesis H from the scheme $\mathcal{C}$ in Example 1.

subtrees that are abbreviated by vertical dots are very similar to their siblings and therefore omitted in order to keep the tree small.

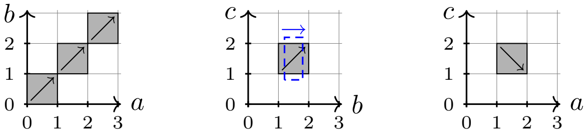
The following theorem then guarantees that $\mathcal{C} \models H$ holds, too.

Theorem 1 (Soundness). *Let $\mathcal{C}$ be an influence scheme and S be a statement. If $\mathcal{C} \vdash S$ then $\mathcal{C} \models S$.*

Proof. First we observe that all the rules are sound in the sense that if $\mathcal{C} \models T$ for all premises T of some rule, then $\mathcal{C} \models S$ for its conclusion S. This is trivial for rule (F) and can be easily be shown by contradiction for the other 11 rules. The theorem can then easily be shown by induction on the height of a proof tree for $\mathcal{C} \vdash S$. □

4 Completeness for Elementary Diamond-Free Schemes

General Incompleteness. We remark that the calculus of influence is not complete in general. Consider the variable order $a < b < c$ and the scheme $\mathcal{C}$ (in grey) and hypothesis H (in dashed blue) represented by the following rectangles.



It seems that H does not follow from $\mathcal{C}$ because it demands constant behaviour of an influence $\mathcal{F}_{b,c}$ on the interval $[1, 2]$ while $\mathcal{C}$ only prescribes monotonic behaviour there. However, we have $\mathcal{C} \models H$ indeed for the following reason: the combination of $S_1 = a \xhookrightarrow{[1,2] \nearrow [1,2]} b$ with $b \xhookrightarrow{[1,2] \nearrow [1,2]} c$ yields $a \xhookrightarrow{[1,2] \nearrow [1,2]} c$. Together with $a \xhookrightarrow{[1,2] \searrow [1,2]} c$ we get $a \xhookrightarrow{[1,2] \rightarrow [1,2]} c$, i.e.

we must have that $\mathcal{F}_{a,c}$ is constant on $[1,2]$ for any $\mathcal{F}$ with $\mathcal{F} \models \mathcal{C}$. Since $\mathcal{F}_{a,c} = \mathcal{F}_{b,c} \circ \mathcal{F}_{a,b}$ and $\mathcal{F}_{a,b}$ cannot be constant on $[1,2]$ because of the two statements neighbouring S_1, we must indeed have that $\mathcal{F}_{b,c}$ is constant on $[1,2]$. Thus, $\mathcal{C} \models H$ but the rules do not support this kind of *backwards* reasoning (from (a,c) to (b,c)). Hence, we have $\mathcal{C} \not\vdash H$.

There are two principal ways to go from here: either extend the calculus by rules formalising this kind of reasoning, or try to achieve completeness for a restricted class of schemes and hypotheses only. We do the latter; the former would require a significant extension of the machinery as the example above shows: backwards reasoning introduces nondeterminism, and in order to resolve it one needs to take contexts of statements into account. This suggests that general completeness may only be achieved through a general extension of the format of rules. Note also that completeness cannot hold for a class of schemes containing inconsistent ones, where $\mathcal{C}$ is said to be *consistent* if there is some $\mathcal{F}$ s.t. $\mathcal{F} \models \mathcal{C}$. The reason is that we have $\mathcal{C} \models H$ for any H whenever $\mathcal{C}$ is inconsistent, even when H makes an assertion about variables not occurring in $\mathcal{C}$ in which case it is clear that H cannot be derived from $\mathcal{C}$.

Normalisation. We develop some general machinery that is useful for obtaining completeness in a restricted case. For a scheme $\mathcal{C}$ and variables a, b with $a < b$ we write $\mathcal{C}_{a,b}$ for the set of statements $S \in \mathcal{C}$ s.t. $S = a \xhookrightarrow{I\,q\,I'} b$ for some I, q, I'.

Definition 2. We call a scheme $\mathcal{C}$ *separated* if for all $a, b \in \mathcal{V}$ with $a < b$ there are $n \in \mathbb{N}$ and $x_1 < \ldots < x_{n+1} \in \mathbb{Q} \cup \{-\infty, \infty\}$, behaviours $q_1, \ldots, q_n$ and intervals $[l_1, u_1], \ldots, [l_n, u_n]$ s.t.

$$\mathcal{C}_{a,b} = \{ \xhookrightarrow{[x_1,x_2]\; q_1\; [l_1,u_1]}, \xhookrightarrow{[x_2,x_3]\; q_2\; [l_2,u_2]}, \ldots, \xhookrightarrow{[x_n,x_{n+1}]\; q_n\; [l_n,u_n]} \} .$$

This induces a natural notion of *left* and *right neighbour* of a statement T in a separated scheme, denoted *lnb*(T) and *rnb*(T) when they exist.

We say that such a separated $\mathcal{C}$ is *minimal* if for all $i = 1, \ldots, n$ we have

a) if $q_i = \nearrow$ then $u_i \leq u_{i+1}$ and $l_{i-1} \leq l_i$,
b) if $q_i = \searrow$ then $l_i \geq l_{i+1}$ and $u_{i-1} \geq u_i$i
c) if $q_i = \rightarrow$ then $u_i \leq \min(u_{i-1}, u_{i+1})$ and $l_i \geq \max(l_{i-1}, l_{i+1})$,

where we set $l_0 = l_{n+1} := -\infty$ and $u_0 = u_{n+1} := \infty$ to avoid case distinctions.

$\mathcal{C}$ is called *transitive* if for all $a, b, c \in \mathcal{V}$ with $a < b < c$ and all $x, y \in \mathbb{R}$ we have the following: if $x \in I_1$, $y \in I_2$ for some statement $a \xhookrightarrow{I_1\, q_1\, I_2} b \in \mathcal{C}$, and $y \in I_3$ for some statement $b \xhookrightarrow{I_3\, q_2\, I_4} c \in \mathcal{C}$, then there is a statement $a \xhookrightarrow{I_5\, q_3\, I_6} c \in \mathcal{C}$ s.t. $x \in I_5$ and $I_6 \subseteq I_4$.

$\mathcal{C}$ is called *normalised* if it is separated, minimal and transitive.

So, intuitively, separation and minimality predict that the statements in a normalised scheme can be arranged as a sequence of horizontally adjacent rectangles, for each pair of variables a, b, with no gaps in between, and no statement can be strengthened further because of its left or right neighbours (compare this

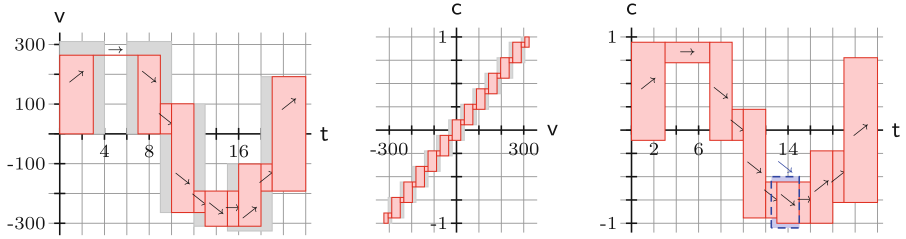


Fig. 5. A normalisation $\mathcal{C}^*$ (red) of the influence scheme $\mathcal{C}$ from Example 1 (Color figure online) (grey).

to the strengthening rules $(\mathsf{L}^+_{\nearrow})$–$(\mathsf{R}^+_{\searrow})$. Transitivity means that $\mathcal{C}$ is complete in the sense that whenever it allows $\mathcal{F}_{a,b}(x) = y$ and $\mathcal{F}_{b,c}(y) = z$ for some x, y, z, then it must also predict the possibility of $\mathcal{F}_{a,c}(x) = z$.

Lemma 1 (Normalisation Lemma). *Let $\mathcal{C}$ be a consistent scheme. There is a normalised scheme $\mathcal{C}^*$ s.t. $\mathcal{C}^* \equiv \mathcal{C}$ and for all $T \in \mathcal{C}^*$ we have $\mathcal{C} \vdash T$.*

Proof. (Sketch) We successively transform $\mathcal{C}$ into $\mathcal{C}^*$ using operations that follow rule applications. (G), (I^+) and (I^-) (in restricted form) can be used to obtain separation, $(\mathsf{L}^+_{\nearrow})$–$(\mathsf{R}^+_{\searrow})$ to ensure minimality, and (T) together with (J) to ensure transitivity. The trick is then to arrange the process of saturating $\mathcal{C}$ by adding new statements and replacing some with others in a terminating way.

In the following, we will write $\mathcal{C}^*$ to denote a normalised scheme obtained from $\mathcal{C}$ that satisfies the conditions of this lemma. Note that $\mathcal{C}^*$ is not necessarily unique; for example statements with adjacent domains and equal ranges and behaviours can be merged using rule (J) or statements can be split w.r.t. to their domain using (I^-) without breaking the conditions of the lemma.

Example 4. Figure 5 shows the result of normalising the scheme $\mathcal{C}$ from Example 1 (grey rectangles) as a scheme $\mathcal{C}^*$ with 11+25+11=47 statements shown as red rectangles. It should be clear that the hypothesis H, also depicted here as a blue rectangle, does indeed follow from $\mathcal{C}^*$: intuitively, it is impossible to draw an influence experiment into these diagrams as three functions that traverse through the red rectangles in the prescribed ways without also traversing through the blue rectangle correctly.

Figure 5 suggests the use of the normalisation process for proof construction: a close inspection of the example proof in Fig. 4 allows the origin of the red rectangles touched by the hypothesis H to be traced back to the grey ones from the original scheme.

Countermodel Construction. The following two lemmas contain one of the main ingredients for obtaining a completeness result: they show how to construct influences on a particular statement in a normalised scheme piecewise to one that satisfies all the statements for the same variables in this scheme. Note that this does not construct an influence experiment (yet) as it does not show how to construct influences for other pairs of variables.

We first make an observation about the possibility to satisfy statements in a normalised scheme by particular influences. A sequence $S_1, \dots, S_m$ of statements $S_i = a \xhookrightarrow{[x_i,y_i]\, q_i\, [x'_i,y'_i]} b$ is called *connected* if $y_i = x_{i+1}$, i.e. $S_{i+1} = rnb(S_i)$ for all $i < n$. A *connector* for $S_1, \dots, S_n$ is an influence f s.t. $dom(f) = [x_1, y_n]$ and, for all $i \leq n$, we have that $f \models \xhookrightarrow{[x_i,y_i]\, q_i\, [x'_i,y'_i]}$. Such a connector f is *strict* if, additionally, for all $i \leq n$ we have $f \not\models \xhookrightarrow{[x_i,y_i]\, q'\, [x'_i,y'_i]}$ for any $q' \prec q_i$. It is *range-covering* if there are $x, y \in [x_1, y_n]$ such that $f(x) = \min\{x'_1, \dots, x'_n\}$ and $f(y) = \max\{y'_1, \dots, y'_n\}$. Sometimes, we will need to construct connectors for single statements S which are simply sequences of length 1 only.

Lemma 2 (Connectors Lemma). *Let $\mathcal{C}$ be consistent and normalised and $S = a \xhookrightarrow{[x,x']\, q\, [y,y']} b \in M$.*

a) Suppose $x'', y'' \in \mathbb{R}$ are given s.t. $x < x'' < x'$ and $y \leq y'' \leq y'$. Then there is a connector f for S s.t. $f(x'') = y''$.
b) Suppose $y'' \in rng(lnb(S)) \cap rng(S)$ is given. Then there is a connector f for S s.t. $f(x) = y''$.
c) Suppose $y'' \in rng(S) \cap rng(rnb(S))$ is given. Then there is a connector f for S s.t. $f(x') = y''$.
d) Let $S_1, \dots, S_n$ be connected s.t. the behaviour of S_i is not $\rightarrow$ for some i. Then there is a strict, range-covering connector for $S_1, \dots, S_n$.

Proof. (Sketch) Parts (a)–(c) essentially boil down to a case distinction, depending on the behaviour q. However, it is relatively easy to observe that the requirements in all three cases are always satisfiable by a function that is either linear or composed of two linear functions on the interval $[x, x']$, making use of the intuitive fact that in a rectangle, with two points given on the left and right edge and one in the middle, it is always possible to draw a (straight) line within this rectangle from the left point to the middle one, and then continue it to the right one. Part (d) requires a decomposition of the sequence $S_1, \dots, S_n$ according to their behaviours.

An immediate consequence of this is the possibility to build influences for not just a single statement in a normalised scheme, but in fact for all the statements concerning the same pair of variables. This crucially relies on parts (b) and (c) of Lemma 2.

Lemma 3 (Small Extension Lemma). *Let $\mathcal{V}$ be a partially ordered set of variables, $a, b \in \mathcal{V}$ s.t. $a < b$, and $\mathcal{C}$ be a consistent and normalised $\mathcal{V}$-influence scheme s.t.*

$$\mathcal{C}_{a,b} = \{\underbrace{\xhookrightarrow{[x_1,x_2]\, q_1\, I_1}}_{T_1}, \underbrace{\xhookrightarrow{[x_2,x_3]\, q_2\, I_2}}_{T_2}, \dots, \underbrace{\xhookrightarrow{[x_n,x_{n+1}]\, q_n\, I_n}}_{T_n}\} .$$

Let $1 \leq j \leq k \leq n$ and f' be a connector for $T_j, \dots, T_k$. Then there is an influence f s.t. $dom(f) = [x_1, x_{n+1}]$, $f \models T_j$ for all $j = 1, \dots, n$, and $f(x) = f'(x)$ for all $x \in [x_j, x_{k+1}]$.

Completeness for Elementary Schemes over Diamond-Free Orders. Let $\mathfrak{C}$ be a class of pairs of schemes and statements. We say that the calculus of influence is *complete for* $\mathfrak{C}$ if for all $(\mathcal{C}, S) \in \mathfrak{C}$ we have: if $\mathcal{C} \models S$ then $\mathcal{C} \vdash S$. We now concentrate on a class that allows for a construction proving completeness, and which still captures a large class of experiments and hypotheses occurring in natural sciences, cf. the concluding section for a discussion on that.

We call a pair (a, b) of variables *elementary* if $a < b$ and there is no c s.t. $a < c < b$. Any finite partial order is the (reflexive-)transitive closure of a finite set of elementary pairs. A statement $a \xhookrightarrow{I\,q\,I'} b$ is called *elementary* if (a, b) is elementary. A scheme $\mathcal{C}$ is called *elementary* if all $T \in \mathcal{C}$ are elementary.

We say that the partial order $\leq$ is *diamond-free* if for all a, b, c, d: if $a \leq b \leq d$ and $a \leq c \leq d$ then $b \leq c$ or $c \leq b$. In a finite diamond-free partial order, for every pair (a, b) with $a < b$ there is a unique sequence $c_1, \ldots, c_n$ for some $n \geq 0$ s.t. $(a, c_1), (c_n, b)$ and (c_i, c_{i+1}) for $i = 1, \ldots, n-1$ are all elementary.

In a diamond-free elementary scheme, all *derivable* non-elementary statements can be traced back to applications of the transitivity rule (T). Moreover, in any normalisation of a diamond-free elementary scheme obtained as in Lemma 1, all non-elementary statements can be traced back to an application of rule (T).

Lemma 4 (Decomposition Lemma). *Let $\mathcal{C}$ be an elementary scheme over a diamond-free partial order and $\mathcal{C}^*$ be a normalisation of $\mathcal{C}$ obtained via Lemma 1. Suppose $T = a \xhookrightarrow{I\,q\,I'} c \in \mathcal{C}^*$ such that (a, c) is non-elementary. Then there is b with $a < b < c$ and $S = a \xhookrightarrow{I\,q_1\,I_1} b$ and $S'_1, \ldots, S'_n$ such that $S, S'_1, \ldots, S'_n \in \mathcal{C}^*$, joining $S'_1, \ldots, S'_n$ via (J) yields $S' = b \xhookrightarrow{I_2\,q_1\,I'} c$, and $q = q_1 \otimes q_2$, $I_1 \subseteq I_2$.*

The key ingredients are that all non-elementary statements in $\mathcal{C}^*$ are derivable in $\mathcal{C}$, and the fact that $\mathcal{C}^*$ is normalised, whence a derivation of T in $\mathcal{C}$ can be used to generate a derivation of T in $\mathcal{C}^*$. Note that w.l.o.g. we can assume that $I_1 = I_2$ in the above lemma.

Now let $\mathcal{C}$ be an elementary diamond-free scheme. We observe that any influence experiment that satisfies all statements in $\mathcal{C}$ on elementary relations automatically satisfies all *derivable* statements on non-elementary relations due to correctness of the rules in the calculus of influence, in particular their observance of the coherence principle. This yields the following.

Lemma 5 (Sufficiency Lemma). *Let $\mathcal{C}$ be an elementary and diamond-free scheme, and let $\mathcal{C}^*$ be a normalisation of $\mathcal{C}$ obtained via Lemma 1. Then any influence experiment that satisfies all elementary statements in $\mathcal{C}^*$ satisfies* all *statements of $\mathcal{C}^*$.*

The next lemma then contains the heart of the completeness proof. It shows how to construct counterexamples, in the form of specific influence experiments, for normalised schemes and hypotheses that appear to state something different to what is contained in the normalised scheme.

Lemma 6 (Counterexample Lemma). *Let $\mathcal{C}$ be a consistent, elementary scheme over a diamond-free partial order and $\mathcal{C}^*$ be a normalisation of $\mathcal{C}$ obtained* *via Lemma 1. Let $a, b \in \mathcal{V}$ s.t. $a < b$ and*

$$\mathcal{C}^*_{a,b} = \{\underbrace{\xhookrightarrow{[x_1,x_2]\; q_1\; [l_1,u_1]}}_{T_1}, \underbrace{\xhookrightarrow{[x_2,x_3]\; q_2\; [l_2,u_2]}}_{T_2}, \dots, \underbrace{\xhookrightarrow{[x_n,x_{n+1}]\; q_n\; [l_n,u_n]}}_{T_n}\} \ .$$

Let $H = a \xhookrightarrow{[x_0,y_0]\; q\; [l,u]} b$. If one of the following conditions holds, then there is an influence experiment $\mathcal{F}$ s.t. $\mathcal{F} \models \mathcal{C}^*$ but $\mathcal{F} \not\models H$.

a) $x_0 < x_1$ or $y_0 > x_{n+1}$.
b) (I) $\bigcup_{h=i}^{j}[l_h, u_h] \not\subseteq [l, u]$ or (II) $\sup_{\preceq}(q_i, \dots, q_j) \not\preceq q$ holds, where i and j are the (necessarily unique) indices s.t. $x_0 \in [x_i, x_{i+1}]$ and $y_0 \in [x_j, x_{j+1}]$.

Proof. (Sketch) We give a high-level, intuitive idea of the construction. If (a, b) is elementary, it suffices to find an $\mathcal{F}_{a,b}$ such that $\mathcal{F}_{a,b} \models \mathcal{C}^*_{a,b}$ but $\mathcal{F}_{a,b} \not\models J$. The functions for the other elementary relations can be interpreted in an arbitrary fashion such that $\mathcal{F}_{c,d}$ satisfies $\mathcal{C}^*_{c,d}$ for all (c, d). This is always possible since $\mathcal{C}$, and hence $\mathcal{C}^*$ is consistent. The interpretations of the non-elementary relations are then obtained automatically via the coherence principle; note that this always satisfies any statements on the respective non-elementary relations due to Lemma 5.

Case (a) is the simpler one. Here, $[x_0, y_0] \subsetneq [x_1, x_{n+1}]$. Hence, it suffices to construct an experiment $\mathcal{F}$ s.t. $dom(\mathcal{F}_{a,b}) = [x_1, x_{n+1}]$, whence $\mathcal{F} \not\models H$. We need to ensure $\mathcal{F} \models \mathcal{C}$ by simply truncating the domain of any influence experiment that satisfies $\mathcal{C}$. Such an experiment exists since $\mathcal{C}$ is consistent.

For case (b), H disagrees with the statements in $\mathcal{C}^*_{a,b}$ in at least one of two ways: (I) it restricts the values of an experiment at some point x more than the unique statement T_i in the sequence in $\mathcal{C}^*_{a,b}$ covering x does. Then we pick a value y that is covered by the vertical interval in T_i but not in H, use Lemma 2 (a) to obtain a connector that runs through this point (x, y) and extend it to an influence using Lemma 3 to ensure $\mathcal{F} \models \mathcal{C}$ but $\mathcal{F} \not\models H$. Or (II) the behaviour stated in H is strictly stronger than those in the corresponding statements in $\mathcal{C}^*_{a,b}$. Then we obtain a strict connector for these statements using Lemma 2 (d) and extend it accordingly using Lemma 3. Strictness ensures that the influence $\mathcal{F}_{a,b}$ has the behaviours required by $\mathcal{C}^*$ but not by H, hence $\mathcal{F} \not\models H$ as well.

If (a, b) is not elementary, by the decomposition lemma (Lemma 4) there is a sequence $a = c_1, \dots, c_n = b$ of elementary relations and a sequence $S_1, \dots, S_{n-1}$ of statements derivable in $\mathcal{C}^*$ that satisfy the requirements of Lemma 4. We omit case (a). If we are in case (b) (I), again we pick a point (x, y) not covered by H, but by the statements in $\mathcal{C}^*_{a,b}$. We then generate a sequence of points (x_i, y_i) for $i \leq n$ such that $x = x_1$ and $y_i = x_{i+1}$ for all $i < n$ and $y_n = y$. It then suffices to invoke Lemma 2 (a) and Lemma 3 to complete the individual relations $\mathcal{F}_{c_i,c_{i+1}}$ such that they go through the point (x_i, y_i).

For the case (b) (II), it suffices to build interpretations of the $\mathcal{F}_{c_i,c_{i+1}}$ that are strict w.r.t. S_i. However, for $i > 1$, the statement T_i might not exist in $\mathcal{C}^*$, but may only be derivable via (J). We use Lemma 2 (d) to obtain a strict, range-covering connector for the sequence of statements that derive S_i and, again, use Lemma 3 to complete it into an influence for $\mathcal{F}_{c_i,c_{i+1}}$. Since these connectors are range-covering, we obtain a strict interpretation for $\mathcal{F}_{a,b}$ from these intermediate $\mathcal{F}_{c_i,c_{i+1}}$, which is the desired contradiction. □

Theorem 2 (Completeness for Elementary Diamond-Free Schemes). *The calculus of influence is complete for the class of consistent and elementary schemes over diamond-free partial orders, and arbitrary hypotheses.*

Proof. Let $\mathcal{C}$ be consistent and elementary, its underlying partial order $\leq$ be diamond-free. Let $\mathcal{C}^*$ be a normalisation of $\mathcal{C}$ obtained via Lemma 1. Hence, $\mathcal{C}^*$ is also consistent. Let $H = a \xhookrightarrow{[x,y]\,q\,I} b$ s.t. $a < b$ and suppose that

$$\mathcal{C}^*_{a,b} = \{\underbrace{\xhookrightarrow{[x_1,x_2]\,q_1\,I_1}}_{T_1}, \underbrace{\xhookrightarrow{[x_2,x_3]\,q_2\,I_2}}_{T_2}, \dots, \underbrace{\xhookrightarrow{[x_n,x_{n+1}]\,q_n\,I_n}}_{T_n}\}\,.$$

Moreover, by Lemma 1 we have $\mathcal{C} \vdash T_i$ for all $i = 1, \dots, n$.

If $x < x_1$ or $y > x_{n+1}$ then Lemma 6 (a) would yield a contradiction to the assumption that $\mathcal{C}^* \models H$. Thus, there are i and j s.t. $x \in [x_i, x_{i+1}]$ and $y \in [x_j, x_{j+1}]$. Now we must have $\bigcup_{h=i}^{j} I_h \subseteq I$ and $\sup_{\preceq}(q_i, \dots, q_j) \preceq q$ for otherwise Lemma 6 (b) would yield a contradiction to the assumption that $\mathcal{C}^* \models H$.

Let $T := a \xhookrightarrow{[x_i,x_{j+1}]\ \sup_{\preceq}(q_i,\dots,q_j)\ I_i \cup \dots \cup I_j} b$. By repeated applications of rule (J), T is provable from $T_i, \dots, T_j$, whence $\mathcal{C} \vdash T$. Moreover, H is provable from T by at most one application of rule (I^-) and (Q^-) each. So $\mathcal{C} \vdash H$ as well. □

The completeness proof shows that for any consistent scheme there is always a satisfying experiment that is comprised of stepwise linear functions. One may argue that this does not capture the heart of functional behaviour in natural sciences. It is possible, though, to require influences not only to be continuous but even differentiable (on their domains). To fulfil this requirement, one could simply use splines of order 3 in the proof of Lemma 2 with their first derivative being 0 at the left and right edges of each rectangle.

5 Proof Search and Empirical Results

We observe that the consequence relation $\vdash$ between influence schemes and hypotheses is in fact polynomial-time decidable, using a bottom-up approach.

Theorem 3. *The problem of deciding, given a scheme $\mathcal{C}$ and a hypothesis H, whether or not $\mathcal{C} \vdash H$ holds, is decidable in time $|\mathcal{C}|^{\mathcal{O}(1)}$.*

Proof. A close inspection of the proof rules shows that rule (I^-) can always be pushed downwards in a proof and successive applications of it can be shortened to a single one, s.t. $C \vdash H$ iff there is some H' which is provable from $\mathcal{C}$ without using rule (I^-), but H can be derived from H' by a single application of (I^-).

Next we observe that all rules except (I^-) have the following property: the bounds of domain and range of the conclusion are bounds of the domain or range of some premise. This guarantees termination of a simple bottom-up procedure for proof search: saturate $\mathcal{C}$ by applications of all rules other than (I^-). The number of different statements created this way is bounded by $4 \cdot v^2 \cdot b^4 = \mathcal{O}(|\mathcal{C}|^6)$ where v is the number of variables occurring in $\mathcal{C}$, and b is the number of different interval bounds occurring in it. For each of these statements, check whether H can be derived using (I^-). This can be done in time polynomial in $|\mathcal{C}|$. □

An implementation of a proof search tool, written in Python, is publicly available.[1] The repository also contains formalisations of some influence schemes and examples of statements whose derivability can be checked using the tool. A deeper look at the implementation details is beyond the scope of this paper and deferred for space considerations. It uses a more sophisticated top-down proof search that constructs only the relevant part of the normalisation of a scheme, i.e. only "around" those statements that can occur in a proof for the given hypothesis H. This can not only contain statements about other variables due to rule (T) but also statements further away from H because rules $(\mathsf{L}^+_{\nearrow})$–$(\mathsf{R}^+_{\searrow})$ can transmit requirements on underlying influence experiments along the horizontal axis.

6 Conclusion

We presented a simple language for statements about the behaviour of functions in a collection that can be interpreted as a way that different entities influence one another. We gave it a formal semantics and devised a proof calculus to characterise the (uncountable) notion of logical consequence that is generally sound and complete for a large class of schemes that covers typical cases occurring in the formal modelling of experimental setups from natural science classes.

It remains to be seen whether the calculus can be extended logically (by further rules for instance) to completely capture a larger class of influence schemes.

Future work will also comprise a number of extensions of the calculus for the purpose of obtaining higher expressiveness. Some experimental setups are inherently temporal in the sense that the influence which a asserts on b depends on a value range of a and a point in time, as in "*Yeast grows at temperatures between 15 and 40° during the next five minutes.*" We have made a proposal to incorporate time in [4]. It also incorporates the ability to make refined assertions about the behaviour of an influence, as in "*Voltage increase is at most* $65.4\ V msec^{-1}$." This replaces the abstract behaviours $\nearrow$ etc. by intervals like $[0, 65.4]$, and the geometric interpretation of a statement becomes a trapezoid.

Formal statements could also include a third interval denoting time points, and influence experiments become collections of binary real-valued functions which interpret cuboids in three-dimensional real spaces. This would also be an approach to model the combined effect of several variables on another variable, even if the modeling of time as a special variable is not desired.

Acknowledgement. We thank Shahla Rasulzade for discussions that have led to this work, and for suggesting to study a temporal extension thereof.

[1] https://github.com/SoerenMoeller/influence_solver.

References

1. Alur, R., et al.: Hybrid modeling and simulation of biomolecular networks. In: Proceedings of 4th International Workshop on Hybrid Systems: Computation and Control, HSCC 2001, vol. 2034, pp. 19–32 (2001)
2. Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof, 2nd edn. Kluwer Academic Publishers, Alphen aan den Rijn (2002)
3. Bortolussi, L., Policriti, A.: Hybrid systems and biology. In: Bernardo, M., Degano, P., Zavattaro, G. (eds.) SFM 2008. LNCS, vol. 5016, pp. 424–448. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68894-5_12
4. Bruse, F., Lange, M., Möller, S.: The calculus of temporal influence. In: Submitted to the 30th International Symposium on Temporal Representation and Reasoning, TIME-2023 (2023)
5. Buss, S.R.: First-order proof theory of arithmetic. In: Handbook of Proof Theory, pp. 79–147. Elsevier, Amsterdam (1998)
6. Chaouiya, C.: Petri net modelling of biological networks. Brief. Bioinf. **8**(4), 210–219 (2007)
7. Ebbinghaus, H.D., Flum, J., Thomas, W.: Mathematical Logic. Undergraduate Texts in Mathematics, 2nd edn. Springer-Verlag, Heidelberg (1994). https://doi.org/10.1007/978-1-4757-2355-7
8. Garg, D., Genovese, V., Negri, S.: Countermodels from sequent calculi in multi-modal logics. In: Proceedings of 27th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2012, pp. 315–324. IEEE (2012)
9. Henkin, L.: The discovery of my completeness proofs. Bull. Symb. Logic **2**(2), 127–158 (1996)
10. Hillmayr, D., Ziernwald, L., Reinhold, F., Hofer, S., Reiss, K.: The potential of digital tools to enhance mathematics and science learning in secondary schools: a context-specific meta-analysis. Comput. Educ. **153**, 103897 (2020)
11. Kastaun, M., Meier, M., Hundeshagen, N., Lange, M.: ProfiLL: professionalisierung durch intelligente Lehr-Lernsysteme. In: Bildung, Schule, Digitalisierung, pp. 357–363. Waxmann-Verlag (2020)
12. Koch, I.: Petri nets - a mathematical formalism to analyze chemical reaction networks. Molec. Inf. **29**(12), 838–843 (2010). https://doi.org/10.1002/minf.201000086
13. Kröger, F.: Temporal Logic of Programs, 1st edn. Springer, Heidelberg (1987). https://doi.org/10.1007/978-3-642-71549-5
14. Negri, S.: Proofs and countermodels in non-classical logics. Logica Universalis **8**(1), 25–60 (2014). https://doi.org/10.1007/s11787-014-0097-1
15. Rademaker, A.: A Proof Theory for Description Logics. Springer Briefs in Computer Science. Springer, Heidelberg (2012)
16. Rathjen, M., Sieg, W.: Proof theory. In: The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University (2020)
17. Smullyan, R.M.: First-Order-Logic, 2 edn. Springer, Heidelberg (1968)
18. Sumatokhin, S., Petrova, O., Serovayskaya, D., Chistiakov, F.: Digitalization of school biological education: problems and solutions. In: SHS Web of Conferences, vol. 79, p. 01016. EDP Sciences (2020)
19. Theocharopoulou, G., Bobori, C., Vlamos, P.: Formal models of biological systems. In: Vlamos, P. (ed.) GeNeDis 2016. AEMB, vol. 988, pp. 325–338. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56246-9_27

A Theory of Cartesian Arrays (with Applications in Quantum Circuit Verification)

Yu-Fang Chen[1], Philipp Rümmer[2,3](✉), and Wei-Lun Tsai[1]

[1] IIS, Academia Sinica, Taipei, Taiwan
[2] University of Regensburg, Regensburg, Germany
ph_r@gmx.net
[3] Uppsala University, Uppsala, Sweden

Abstract. We present a theory of Cartesian arrays, which are multi-dimensional arrays with support for the projection of arrays to sub-arrays, as well as for updating sub-arrays. The resulting logic is an extension of Combinatorial Array Logic (CAL) and is motivated by the analysis of quantum circuits: using projection, we can succinctly encode the semantics of quantum gates as quantifier-free formulas and verify the end-to-end correctness of quantum circuits. Since the logic is expressive enough to represent quantum circuits succinctly, it necessarily has a high complexity; as we show, it suffices to encode the k-color problem of a graph under a succinct circuit representation, an NEXPTIME-complete problem. We present an NEXPTIME decision procedure for the logic and report on preliminary experiments with the analysis of quantum circuits using this decision procedure.

1 Introduction

There has been extensive research on logics to reason about array data-types in programs. Arrays can concisely represent the values of an unbounded number of memory locations, and have been successfully applied to verify industrial-scale programs [11,15,29]. An array formula encoding the semantics of a program path is typically linear in the number of program statements. Much of the existing work focuses on one-dimensional arrays and uses nesting to handle the case of multiple dimensions.

$\|00000\rangle$	69%
$\|00001\rangle$	1%
$\dots$	
$\|11111\rangle$	1%

Fig. 1. A quantum state.

This paper studies a logic called *Cartesian Array Logic (CaAL)*, in which multi-dimensional arrays are treated as first-class citizens. The motivation for designing this logic comes from developing a tailor-made theory for reasoning about *quantum circuits* or *programs,* which need a fundamentally different representation of states than classical programs. *Quantum states* exist in a *superposition* of classical states. Figure 1 gives an example of a 5-qubit quantum state,

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 170–189, 2023.
https://doi.org/10.1007/978-3-031-38499-8_10

which can be interpreted as a probability distribution over 2^5 classical states; every classical state, which can be seen as a string of n bits, is associated with a probability of being observed.

Current SMT-based solutions for reasoning about quantum programs [3] encode program paths to a Satisfiability Modulo Theories (SMT) formula over the theory of real numbers. For a n-qubit quantum program, the direct encoding uses 2^n variables to represent the execution of a quantum circuit, one variable per classical state. The formula representing a quantum circuit is exponential in the circuit size.

In the Cartesian Array Logic designed in this paper, one can instead encode an n-qubit quantum state as an array $s : (\mathbb{B}^n \Rightarrow \mathbb{C})$ that maps each classical state to a complex number c encoding the probability of this classical state being observed. The squared absolute value $|c|^2$ is the probability that the complex number c encodes. *Quantum gates,* the basic operating units of a *quantum circuit,* can be viewed as functions that transform one quantum state to another. We show that CaAL can concisely encode the semantics of quantum gates, so that a path formula becomes linear in the circuit size. The semantics of a quantum circuit is the composition of the gate encodings.

Structure of the Paper. The syntax and formal semantics of the CaAL logic will be given in Sect. 2. In the same section, we show that this logic is quite expressive, it can easily encode the satisfiability problem of a quantified Boolean formula (QBF). We show that deciding the logic is, in fact, NEXPTIME-hard by a polynomial reduction from the k-color problem of a succinct circuit representation of graphs [23]. As an application, in Sect. 3, we show that the logic can concisely encode the semantics of *quantum circuits*, using $\mathbb{B}^n$ as the index type and $\mathbb{C}$ as the value type. In Sect. 4, we present a decision procedure for CaAL, extending the classical approach of read-over-write propagation used for arrays. In the worst case, our procedure might perform an exponential number of such propagations; hence, if the underlying logic can be decided in NP, our logic can be decided in NEXPTIME. The preliminary experimental results of applying this decision for quantum circuit verification can be found in Sect. 5.

Contributions of the paper are (i) a new array logic, CaAL, with native support for multi-dimensional arrays; (ii) the proof the satisfiability problem of CaAL is NEXPTIME-hard; (iii) a linear encoding of the semantics of quantum circuits in CaAL; (iv) an NEXPTIME decision procedure for CaAL without nested array sorts; and (v) a preliminary evaluation of our approach using standard quantum circuits.

Related Work on Verification of Quantum Circuits. Although quantum states can be naturally represented as arrays, the connection between array theories and quantum circuit verification is novel, to the best of our knowledge. In the past, people have considered automated quantum circuit verification based on automata [7], various types of equivalence checking [1,9,19,33], abstract interpretation [24,34], and model checking [13,21,32]. However, techniques based on satisfiability modulo theories (SMT) are still lacking. The closest work to ours is a symbolic execution and verification framework of quantum circuits [3]. The work encodes quantum circuit verification problems into SMT with the theory of real numbers, using variables in trigonometric functions, e.g., $\sin x$, which might lose precision in corner cases. As mentioned, their approach requires 2^n variables to encode a n-qubit circuit in the worst case. As far as we know, our work is the first SMT-based approach that allows a precise and succinct encoding and verification of quantum circuits.

Related Work on Array Theories. There is a large body of research on array decision procedures for SMT, going back to the 1980s, and most SMT solvers implement at least the theory of extensional arrays (with operations *read* and *write*/*store*) in our paper, as standardized in SMT-LIB [2]. Stump et al. [29] presented a decision procedure for this theory and formed the basis for many later procedures. An extension of the theory, called Combinatorial Array Logic (CAL), with functions for *constant arrays* and for the *point-wise extension of functions* was presented by De Moura et al. [11]. CAL served as the main inspiration for our work and is in this paper extended further by adding *projections* and *updates of sub-arrays.* An extension of CAL with *cardinality constraints* was presented by Raya et al. [25]. Christ et al. [8] present an algorithm for the theory of arrays where lemmas are created lazily based on weak equivalences; this method was later extended to handle *constant arrays* [20].

There are also many more generalized decision procedures for arrays. For instance, Ganesh et al. [16] focus on the combined theory of arrays and bit-vectors and present a decision procedure based on pre-processing, bit-blasting, and linear arithmetic solving. Brummayer et al. present a decision procedure for the same theory that introduces lemmas lazily, guided by congruence closure [6]. An extended array theory tailored to software, including operations memset and memcpy, was presented by Falke et al. [12]. More recently, several theories of finite arrays were proposed. Bonacina et al. [5] extend the standard theory of arrays with an abstract notion of length, and present a decision procedure based on the CDSAT framework. Wang et al. [31] consider a logic extending CAL with a length function, as well as operations for concatenation, slicing, and repetition of arrays, and identify a decidable fragment. Sheng et al. [27] propose a theory of sequences that combines the standard array operations with a length function, concatenation, and slicing. All those logics cannot directly encode quantum circuits in a similar style as CaAL, however, since no projection operation is available.

2 A Theory of Cartesian Arrays

2.1 Preliminaries

We work in the setting of multi-sorted first-order logic with equality; see, e.g., [18]. A signature is a tuple $\Sigma = (\Sigma^S, \Sigma^F, \Sigma^P)$ consisting of a set Σ^S of sorts, a set Σ^F of function symbols, and a set Σ^P of predicates. Predicates and functions have fixed arity and argument sorts, and functions have a fixed result sort. Given a signature Σ and a set $\mathcal{X}$ of sorted variables, we define the usual notions of Σ-terms, Σ-atoms, Σ-literals, Σ-formulas, and Σ-sentences. Formulas are evaluated over Σ-structures $M = (D, I)$ that interpret every sort $\sigma \in \Sigma^S$ as a non-empty domain $I(\sigma) \subseteq D$, predicates $p \in \Sigma^P$ as relations $I(p)$, and functions $f \in \Sigma^F$ as set-theoretical functions $I(f)$. We slightly abuse notation; we assume that also variables $x \in \mathcal{X}$ are mapped to values $I(x)$ by M. The evaluation of terms, formulas, etc., is defined as is common; the equality symbol $=$ is assumed to be interpreted as the equality relation on D. A theory T over Σ is a set of Σ-sentences. A Σ-formula ϕ is called T-satisfiable if there is a Σ-structure M satisfying both the T-axioms and ϕ.

2.2 Definition of the Theory of Cartesian Arrays

Cartesian arrays are introduced in the context of a base signature Σ_B and a base Σ_B-theory T_B, which provides the index and value sorts for arrays. The signature $\Sigma_{\text{CaAL}} = (\Sigma^S_{\text{CaAL}}, \Sigma^F_{\text{CaAL}}, \Sigma^P_{\text{CaAL}})$ of CaAL is then defined as follows. The set of sorts is the least set Σ^S_{CaAL} such that (i) $\Sigma^S_B \subseteq \Sigma^S_{\text{CaAL}}$, and (ii) $\sigma, \tau \in \Sigma^S_{\text{CaAL}}$ and $n \in \mathbb{N}_{>0}$ imply $(\sigma^n \Rightarrow \tau) \in \Sigma^S_{\text{CaAL}}$. A sort $(\sigma^n \Rightarrow \tau)$ is an array sort of arity n with index sort σ and value sort τ.

Table 1. Operations included in Σ^F_{CaAL} for each sort $(\sigma^n \Rightarrow \tau)$.

$\cdot[\cdot, \ldots, \cdot] : (\sigma^n \Rightarrow \tau) \times \sigma^n \to \tau$	Reading of array values
$\mathit{store} : (\sigma^n \Rightarrow \tau) \times \sigma^n \times \tau \to (\sigma^n \Rightarrow \tau)$	Updating of array values
$K : \tau \to (\sigma^n \Rightarrow \tau)$	Construction of constant arrays
$\mathit{map}_f : (\sigma^n \Rightarrow \tau_1) \times \cdots \times (\sigma^n \Rightarrow \tau_k) \to (\sigma^n \Rightarrow \tau)$	Point-wise extension of base function $f : \tau_1 \times \cdots \times \tau_k \to \tau$
$\mathit{proj}_k : (\sigma^n \Rightarrow \tau) \times \sigma \to (\sigma^{n-1} \Rightarrow \tau)$	For $n > 1$ and $k \in \{1, \ldots, n\}$, projection to $n - 1$ of the indexes
$\mathit{arrayStore}_k : (\sigma^n \Rightarrow \tau) \times \sigma \times (\sigma^{n-1} \Rightarrow \tau) \to (\sigma^n \Rightarrow \tau)$	For $n > 1$ and $k \in \{1, \ldots, n\}$, update of a sub-array

The set Σ^F_{CaAL} includes Σ^S_B, as well as the operations listed in Table 1 for every array sort $(\sigma^n \Rightarrow \tau)$. The operators $\cdot[\cdot, \ldots, \cdot]$ and *store* are the functions for reading from and writing to arrays, as in the standard theory of arrays. K and map_f correspond to the functions introduced in CAL [11]; in particular, any base function $f \in \Sigma^F_B$ is lifted to an operator on arrays using map_f. The operators *proj* and *arrayStore* are specific to our theory CaAL, and can be used to project an n-dimensional array to an $(n-1)$-dimensional sub-array by fixing the value of the k'th index, and to update the corresponding portion of the original array, respectively. The set Σ^P_{CaAL} coincides with Σ^P_B. Semantics is defined by the axiom schemata in Table 2.

Example 1. We illustrate the use of two-dimension arrays $s, s' : (\mathbb{B}^2 \Rightarrow \mathbb{C})$ to encode two-qubit quantum states. Suppose that s represents the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, and $s' = X_2(s)$ is the quantum state after applying an X gate (the quantum version of a "not"-gate) on the 2nd qubit of s. The matrix representations of s and s' are as follows; note that the results of $x_2 = 0$ and $x_2 = 1$ are swapped in s and s'.

$$s = \begin{array}{c} \\ {\scriptstyle x_2=0} \\ {\scriptstyle x_2=1} \end{array} \begin{array}{c} \begin{array}{cc} {\scriptstyle x_1=0} & {\scriptstyle x_1=1} \end{array} \\ \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \end{array}, \quad s' = \begin{array}{c} \\ {\scriptstyle x_2=0} \\ {\scriptstyle x_2=1} \end{array} \begin{array}{c} \begin{array}{cc} {\scriptstyle x_1=0} & {\scriptstyle x_1=1} \end{array} \\ \begin{pmatrix} 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 \end{pmatrix} \end{array}.$$

The projection $\mathit{proj}_1(s, k)$ maps the matrix s to its k'th column vector, specifically the column with $x_1 = k$. In CaAL, we can construct s' from s as $s' = \mathit{arrayStore}_2(\mathit{arrayStore}_2(K(0), 1, \mathit{proj}_2(s, 0)), 0, \mathit{proj}_2(s, 1))$. To compute the sum of the two matrices, we use $\mathit{map}_+(s, s')$, which is also utilized for other quantum gate operations.

Several extensions of the theory of Cartesian arrays are possible but beyond the scope of this paper. Those include (i) arrays with multiple different index sorts, as opposed to just n copies of the same index sort σ; and (ii) a theory that also includes point-wise extensions of predicates.

Table 2. Axioms of the Theory of Cartesian Arrays. As shorthand notation, we write $\bar{i} : \sigma^n$ for a vector of n index variables $i_1 : \sigma, \ldots, i_n : \sigma$.

$$\begin{aligned} &\forall a : (\sigma^n \Rightarrow \tau), \bar{i} : \sigma^n, x : \tau. \\ &\qquad \mathit{store}(a, \bar{i}, x)[\,\bar{i}\,] = x \end{aligned} \tag{1}$$

$$\begin{aligned} &\forall a : (\sigma^n \Rightarrow \tau), \bar{i} : \sigma^n, \bar{j} : \sigma^n, x : \tau. \\ &\qquad \bar{i} = \bar{j} \vee \mathit{store}(a, \bar{i}, x)[\,\bar{j}\,] = a[\,\bar{j}\,] \end{aligned} \tag{2}$$

$$\begin{aligned} &\forall a, b : (\sigma^n \Rightarrow \tau).\ \exists \bar{i} : \sigma^n. \\ &\qquad a = b \vee a[\,\bar{i}\,] \neq b[\,\bar{i}\,] \end{aligned} \tag{3}$$

$$\begin{aligned} &\forall x : \tau, \bar{i} : \sigma^n. \\ &\qquad K(x)[\,\bar{i}\,] = x \end{aligned} \tag{4}$$

$$\begin{aligned} &\forall a_1 : (\sigma^n \Rightarrow \tau_1), \ldots, a_k : (\sigma^n \Rightarrow \tau_k), \bar{i} : \sigma^n. \\ &\qquad \mathit{map}_f(a_1, \ldots, a_k)[\,\bar{i}\,] = f(a_1[\,\bar{i}\,], \ldots, a_k[\,\bar{i}\,]) \end{aligned} \tag{5}$$

$$\begin{aligned} &\forall a : (\sigma^n \Rightarrow \tau), \bar{i} : \sigma^n. \\ &\qquad \mathit{proj}_k(a, i_k)[i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_n] = a[\,\bar{i}\,] \end{aligned} \tag{6}$$

$$\begin{aligned} &\forall a : (\sigma^n \Rightarrow \tau), b : (\sigma^{n-1} \Rightarrow \tau), \bar{i} : \sigma^n. \\ &\qquad \mathit{arrayStore}_k(a, i_k, b)[\,\bar{i}\,] = b[i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_n] \end{aligned} \tag{7}$$

$$\begin{aligned} &\forall a : (\sigma^n \Rightarrow \tau), b : (\sigma^{n-1} \Rightarrow \tau), \bar{i} : \sigma^n, j : \sigma. \\ &\qquad j = i_k \vee \mathit{arrayStore}_k(a, j, b)[\,\bar{i}\,] = a[\,\bar{i}\,] \end{aligned} \tag{8}$$

2.3 Complexity of Satisfiability in CaAL

We now study the hardness of satisfiability of quantifier-free CaAL formulas. The quantified Boolean formula problem (QBF) generalizes the Boolean satisfiability problem by allowing *existential* and *universal* quantifiers to be applied to variables. Its satisfiability problem is *PSPACE-complete* [28]. Without loss of generality, we can assume that QBF formulas are in *prenex normal form* $Q_1 x_1 . Q_2 x_2 . \cdots Q_n x_n . \phi$, which consists of a Boolean formula ϕ over n Boolean variables $x_1, \ldots, x_n$, and a prefix of quantifiers $Q_1, Q_2, \ldots, Q_n \in \{\forall, \exists\}$.

To reduce the satisfiability problem of QBF to CaAL, we assume that the base theory provides a sort $\mathbb{B}$ with the standard operations. This sort will be used for both index and values. An array $\mathsf{toCaAL}(\phi) : (\mathbb{B}^n \Rightarrow \mathbb{B})$ encoding the semantics of ϕ is defined recursively as follows:

- $\mathsf{toCaAL}(x_k) = \mathit{arrayStore}_k(K(0), 1, K(1))$.
- $\mathsf{toCaAL}(\neg\phi) = \mathit{map}_\neg(\mathsf{toCaAL}(\phi))$.
- $\mathsf{toCaAL}(\phi_1 \wedge \phi_2) = \mathit{map}_\wedge(\mathsf{toCaAL}(\phi_1), \mathsf{toCaAL}(\phi_2))$.

Observe that $\mathit{arrayStore}_k(K(0), 1, K(1))[i_1, \ldots, i_k, \ldots, i_n] = i_k$, and note that the size of $\mathsf{toCaAL}(\phi)$ is linear in the size of ϕ. We can construct a CaAL formula that is equisatisfiable with $Q_1 x_1 . \cdots Q_n x_n . \phi$ as follows:

$$\mathsf{QElim}(Q_1 x_1 . \cdots Q_n x_n . \phi) =$$

$$(q_1[0] \odot_1 q_1[1]) \wedge \bigwedge_{i=2}^{n} q_{i-1} = \mathit{map}_{\odot_i}(\mathit{proj}_i(q_i, 0), \mathit{proj}_i(q_i, 1)) \wedge q_n = \mathsf{toCaAL}(\phi)$$

where $\odot_i = \wedge$ when $Q_i = \forall$, and $\odot_i = \vee$ otherwise. Note that the QBF formula $Q_1 x_1 . \cdots Q_n x_n . \phi$ is valid if and only if the CaAL formula $\mathsf{QElim}(Q_1 x_1 . \cdots Q_n x_n . \phi)$ is satisfiable.

Theorem 1. *The satisfiability problem of CaAL over* $\mathbb{B}$ *is PSPACE-hard.*

This lower bound can be improved, however. The k-colorability problem for graphs with succinct circuit representation is NEXPTIME-complete [23]. This problem can be reduced to the satisfiability problem of CaAL in polynomial time as well.

Consider an undirected graph with 2^n nodes, and let $\phi(\bar{x}, \bar{x'})$ be a Boolean circuit encoding the edge relation of the graph: $\phi(\bar{x}, \bar{x'})$ evaluates to true whenever there is an edge $(\bar{x}) \rightarrow (\bar{x'})$ in the graph. The k-colorability of the graph can be characterized as the following formula, where $c : (\mathbb{B}^n \rightarrow \mathbb{N})$ is an array representing the color of each node:

$$\forall \bar{x}, \bar{x'} : \mathbb{B}^n .\ \phi(\bar{x}, \bar{x'}) \rightarrow c[\bar{x}] \neq c[\bar{x'}] \wedge c[\bar{x}] < k \wedge c[\bar{x'}] < k\ .$$

In a similar way as for QBF, we encode ϕ as an array formula ϕ' of linear size, in which $a_\phi : (\mathbb{B}^n \times \mathbb{B}^n \Rightarrow \mathbb{B})$ is an array variable representing the edge relation.

We then create two intermediate arrays $a, b : (\mathbb{B}^n \times \mathbb{B}^n \Rightarrow \mathbb{N})$ and use the following formula in CaAL to encode the relation $\forall \bar{x}, \bar{x'} : \mathbb{B}^n.\ a[\bar{x}, \bar{x'}] = c[\bar{x}] \wedge b[\bar{x}, \bar{x'}] = c[\bar{x'}]$:

$$\begin{aligned}
&\mathsf{EqColor}(a, b, c) \equiv \\
&a = a_n \wedge c = a_0 \ \wedge \ \bigwedge_{j=1}^{n} \mathit{proj}_{j+n}(a_j, 0) = \mathit{proj}_{j+n}(a_j, 1) = a_{j-1} \wedge \\
&b = b_n \wedge c = b_0 \ \wedge \ \bigwedge_{j=1}^{n} \mathit{proj}_{j}(b_j, 0) = \mathit{proj}_{j}(b_j, 1) = b_{j-1}
\end{aligned}$$

Then we encode the k-color problem with the following CaAL formula:

$$\begin{aligned}
&\phi' \wedge \mathsf{EqColor}(a, b, c) \wedge \mathit{map}_f(a_\phi, a, b) = K(1) \\
\text{where}\quad &f(e, \mathit{col1}, \mathit{col2}) \ \equiv \ e \rightarrow (\mathit{col1} \neq \mathit{col2} \wedge \mathit{col1} < k \wedge \mathit{col2} < k).
\end{aligned}$$

Theorem 2. *The satisfiability problem of CaAL is NEXPTIME-hard.*

3 Array Semantics of Quantum Circuits

As an application, we show that CaAL can encode the semantics of quantum circuits. Below, we only give a short overview of quantum circuits and define notations; for more details, see, e.g., the textbook of Nielsen and Chuang [22].

In a n-qubit quantum, a state is a *superposition* of *computational basis states* $\{|j\rangle \mid j \in \{0,1\}^n\}$. For example, for a system with three qubits x_1, x_2, and x_3, the computational basis state $|101\rangle$ (in Dirac notation) denotes a state in which both x_1 and x_3 are set to 1, and x_2 is set to 0. A n-qubit quantum state s is then denoted as a formal sum $\sum_{j \in \{0,1\}^n} c_j \cdot |j\rangle$, where $c_0, c_1, \ldots, c_{2^n-1} \in \mathbb{C}$ are *complex probability amplitudes* satisfying the constraint that $\sum_{j \in \{0,1\}^n} |c_j|^2 = 1$. Intuitively, $|c_j|^2$ is the probability that when we measure the quantum state s in the computational basis, we obtain the basis state $|j\rangle$. The constraint $\sum_{j \in \{0,1\}^n} |c_j|^2 = 1$ states that probabilities need to sum up to 1 for all computational basis states.

We can record a quantum state as an array that maps a computational basis state to its complex probability amplitudes. The state s is represented as an array $s : (\mathbb{B}^n \Rightarrow \mathbb{C})$ satisfying $s[j] = c_j$ for all $j \in \{0,1\}^n$; slightly abusing notation, we denote both the state and the array by s.

3.1 Quantum Circuits

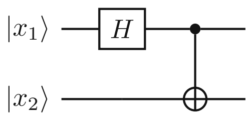


Fig. 2. The EPR circuit, consisting of an H and a CX gate with control qubit ($\bullet$) and target qubit ($\oplus$).

A *quantum circuit* consists of a sequence of *quantum gates.* Each quantum gate defines a specific transformation of quantum states. For example, the Pauli-X gate (the quantum version of classical "not" gate) on the k-th qubit transforms a state s to s' satisfying $\forall i \in \{0,1\}^{k-1}, b \in \{0,1\}, j \in \{0,1\}^{n-k} : s'[ibj] = s[i\bar{b}j]$, i.e., it negates the k-th index bit.

Another example is the Pauli-Z gate on the k-th qubit, which transforms a state s to s' satisfying $\forall i \in \{0,1\}^{k-1}, b \in \{0,1\}, j \in \{0,1\}^{n-k} : s'[ibj] = ite(b, -1 \cdot s[ibj], s[ibj])$. Here, probability amplitudes are multiplied with -1 when b is 1, and are unchanged otherwise.

A H gate, or Hadamard gate, on the k-th qubit transforms a state s to s' satisfying $\forall i \in \{0,1\}^{k-1}, b \in \{0,1\}, j \in \{0,1\}^{n-k}$:

$$s'[ibj] = ite(b, \frac{s[i0j] - s[i1j]}{\sqrt{2}}, \frac{s[i0j] + s[i1j]}{\sqrt{2}}).$$

Notice that the amplitude of a basis state of s' is affected by the amplitude of two basis states of s, enabling a more diverse superposition. The division with $\sqrt{2}$ is for normalizing the probability sum.

A more advanced class of gates are multiple-qubit gates. The CX gate ("controlled-X") on the control qubit c and target qubit t applies an X gate to t when c is 1, and is identity otherwise. Formally, assuming $c < t$, the gate transforms a state s to s' satisfying $\forall i_1 \in \{0,1\}^{c-1}, b_c \in \{0,1\}, i_2 \in \{0,1\}^{t-c-1}, b_t \in \{0,1\}, i_3 \in \{0,1\}^{n-t}$:

$$s'[i_1 b_c i_2 b_t i_3] = ite(b_c, s[i_1 b_c i_2 \bar{b}_t i_3], s[i_1 b_c i_2 b_t i_3]).$$

The Toffoli gate CCX ("controlled-controlled-X gate") has two control qubit c, d and applies the X gate to the target qubit t only when $c = d = 1$.

We have introduced enough quantum gates to define the EPR circuit (Fig. 2), named after Einstein, Podolsky, and Rosen for constructing the Bell state, i.e., a 2-qubit circuit converting a basis state $|00\rangle$ to a maximally entangled state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Starting from a state s (represented s that maps 00 to 1 and others to 0, the circuit first applies H on the first qubit x_1 (denoted H_1 in this paper) to produce the quantum state s' with $s'[00] = s'[10] = \frac{1}{\sqrt{2}}$ and $s'[11] = s'[01] = 0$. Then a $CX_{1,2}$ converts it further to s'' with $s''[00] = s''[11] = \frac{1}{\sqrt{2}}$ and $s''[01] = s''[10] = 0$. Notice that $CX_{1,2}$ converts $|10\rangle$ to $|11\rangle$, i.e., when x_1 is 1, it negates x_2.

Note on Complexity. Simulation of a quantum circuit is bounded-error quantum polynomial time (BQP) hard, a complexity class that is incomparable with NP,

Table 3. Semantics of quantum gates in Cartesian array logic. We use s and s' to denote the quantum state before and after executing the circuit.

Gate	Formula
X_k	$\mathit{proj}_k(s',0) = \mathit{proj}_k(s,1) \wedge$ $\mathit{proj}_k(s',1) = \mathit{proj}_k(s,0)$
Y_k	$\mathit{proj}_k(s',0) = \mathit{map}_{*(-\omega^2)}\mathit{proj}_k(s,1) \wedge$ $\mathit{proj}_k(s',1) = \mathit{map}_{*(\omega^2)}\ \mathit{proj}_k(s,0)$
Z_k	$\mathit{proj}_k(s',0) = \mathit{proj}_k(s,0) \wedge$ $\mathit{proj}_k(s',1) = \mathit{map}_{*(-1)}\mathit{proj}_k(s,1)$
S_k	$\mathit{proj}_k(s',0) = \mathit{proj}_k(s,0) \wedge$ $\mathit{proj}_k(s',1) = \mathit{map}_{*(\omega^2)}\ \mathit{proj}_k(s,1)$
T_k	$\mathit{proj}_k(s',0) = \mathit{proj}_k(s,0) \wedge$ $\mathit{proj}_k(s',1) = \mathit{map}_{*(\omega)}\ \mathit{proj}_k(s,1)$
H_k	$\mathit{proj}_k(s',0) = \mathit{map}_{(.+.)/\sqrt{2}}(\mathit{proj}_k(s,0), \mathit{proj}_k(s,1)) \wedge$ $\mathit{proj}_k(s',1) = \mathit{map}_{(.-.)/\sqrt{2}}(\mathit{proj}_k(s,0), \mathit{proj}_k(s,1))$
$Rx(\frac{\pi}{2})_k$	$\mathit{proj}_k(s',0) = \mathit{map}_{(.+(-\omega^2)*.)/\sqrt{2}}(\mathit{proj}_k(s,0), \mathit{proj}_k(s,1)) \wedge$ $\mathit{proj}_k(s',1) = \mathit{map}_{((-\omega^2)*.+.)/\sqrt{2}}(\mathit{proj}_k(s,0), \mathit{proj}_k(s,1))$
$Ry(\frac{\pi}{2})_k$	$\mathit{proj}_k(s',0) = \mathit{map}_{(.-.)/\sqrt{2}}(\mathit{proj}_k(s,0), \mathit{proj}_k(s,1)) \wedge$ $\mathit{proj}_k(s',1) = \mathit{map}_{(.+.)/\sqrt{2}}(\mathit{proj}_k(s,0), \mathit{proj}_k(s,1))$
$CX_{c,t}$	$\mathit{proj}_c(s',0) = \mathit{proj}_c(s,0) \wedge$ $\mathit{proj}_t(\mathit{proj}_c(s',1),0) = \mathit{proj}_t(\mathit{proj}_c(s,1),1)\wedge$ $\mathit{proj}_t(\mathit{proj}_c(s',1),1) = \mathit{proj}_t(\mathit{proj}_c(s,1),0)$
$CZ_{c,t}$	$\mathit{proj}_c(s',0) = \mathit{proj}_c(s,0) \wedge$ $\mathit{proj}_t(\mathit{proj}_c(s',1),0) = \mathit{proj}_t(\mathit{proj}_c(s,1),0)\wedge$ $\mathit{proj}_t(\mathit{proj}_c(s',1),1) = \mathit{map}_{*(-1)}\mathit{proj}_t(\mathit{proj}_c(s,1),1)$
$CCX_{c,d,t}$	$\mathit{proj}_c(s',0) = \mathit{proj}_c(s,0) \wedge$ $\mathit{proj}_d(s',0) = \mathit{proj}_d(s,0) \wedge$ $\mathit{proj}_t(\mathit{proj}_d(\mathit{proj}_c(s',1),1),0) = \mathit{proj}_t(\mathit{proj}_d(\mathit{proj}_c(s,1),1),1) \wedge$ $\mathit{proj}_t(\mathit{proj}_d(\mathit{proj}_c(s',1),1),1) = \mathit{proj}_t(\mathit{proj}_d(\mathit{proj}_c(s,1),1),0)$

as it can compute exactly the probability amplitudes of a quantum state after executing a circuit. We will show that the Cartesian array logic can encode the semantics of quantum circuits, so one can also use the logic for quantum circuit simulation. Hence, exponential time is the best deterministic algorithm we can hope for when solving CaAL formulas.

3.2 Interpretation of Quantum Gates

We show the encoding of quantum gates in CaAL in Table 3. Notice that this gate set includes several universal gates (e.g., H, CX, and T [10]) that can approximate *any quantum gate* to an arbitrary precision requirement. Arbitrary degree rotation can also be supported using the theory of reals as the base theory. This paper presents a precise encoding that only requires a theory of integers. In the figure, we use s and s' to denote the quantum states (encoded as arrays) before and after executing a quantum gate. To encode $s' = X_k(s)$, negating the k-th qubit, we use $\mathit{proj}_k(s',0) = \mathit{proj}_k(s,1) \wedge \mathit{proj}_k(s',1) = \mathit{proj}_k(s,0)$: index $k = 0$ in s' equals the case of $k = 1$ in s. The handling of Z, S, and T gates is similar, using the *map* function to multiply the array values with different constants. Note that here we use ω to represent $e^{\frac{\pi i}{4}} = \cos\frac{\pi}{4} + i\sin\frac{\pi}{4} = \frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}}$, the unit vector that is at an angle of 45° to the positive real axis in the complex plane. Later we will show that this representation allows a precise algebraic representation of complex numbers using a five-tuple of integers. Observe that $\omega^4 = -1$. The Y gate combines the two constructions; it negates the k-th index qubit and multiplies each projection with different constant coefficients. For the H, $Rx(\frac{\pi}{2})$, and $Ry(\frac{\pi}{2})$ gates, we use a binary *map* function to update the amplitudes. For the controlled gates, we use the projection function to classify the cases according to the control bits and apply the X or Z gate only when all controlled bits are 1.

Example 2. We use CaAL to verify the correctness of the EPR circuit Fig. 2: the circuit transforms the state $|00\rangle$ to $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. For this, the initial state of the circuit is encoded as an array expression, the H and CX gates are encoded according to Table 3, and the intended final state of the circuit is represented as a negated equation:

$$\begin{aligned}
& s_0 = \mathit{store}(K(0),(0,0),1) \\
\wedge\ & \left.\begin{aligned}\mathit{proj}_1(s_1,0) &= \mathit{map}_{(.+.)/\sqrt{2}}(\mathit{proj}_1(s_0,0),\mathit{proj}_1(s_0,1)) \\ \wedge\ \mathit{proj}_1(s_1,1) &= \mathit{map}_{(.-.)/\sqrt{2}}(\mathit{proj}_1(s_0,0),\mathit{proj}_1(s_0,1))\end{aligned}\right\}\ s_1 = H_1(s_0) \\
\wedge\ & \left.\begin{aligned}\mathit{proj}_1(s_2,0) &= \mathit{proj}_1(s_1,0) \\ \wedge\ \mathit{proj}_2(\mathit{proj}_1(s_2,1),0) &= \mathit{proj}_2(\mathit{proj}_1(s_1,1),1) \\ \wedge\ \mathit{proj}_2(\mathit{proj}_1(s_2,1),1) &= \mathit{proj}_2(\mathit{proj}_1(s_1,1),0)\end{aligned}\right\}\ s_2 = CX_{1,2}(s_1) \\
\wedge\ & s_2 \neq \mathit{store}(\mathit{store}(K(0),(1,1),\frac{1}{\sqrt{2}}),(0,0),\frac{1}{\sqrt{2}})
\end{aligned}$$

The formula is unsatisfiable if and only if the EPR circuit correctly performs the transformation.

Representation of Complex Numbers. To achieve accuracy with no loss of precision, in this paper, when working with $\mathbb{C}$, we use a subset of the complex numbers that the following algebraic encoding can express (cf. [7,30,35]):

$$\left(\frac{1}{\sqrt{2}}\right)^k (a + b\omega + c\omega^2 + d\omega^3), \tag{9}$$

Table 4. Tableau proof rules of the decision procedure for CaAL.

$$\mathsf{idx}\ \frac{a = \mathit{store}(b, \bar{i}, v)}{v = a[\bar{i}]} \qquad \mathsf{K}\Downarrow\ \frac{a = K(v) \qquad w = a'[\bar{i}] \qquad a \sim a'}{v = w}$$

$$\mathsf{store}\Downarrow\ \frac{a = \mathit{store}(b, \bar{i}, v) \qquad w = a'[\bar{j}] \qquad a \sim a'}{\bar{i} = \bar{j} \;\;\Big|\;\; w = b[\bar{j}]}$$

$$\mathsf{store}\Uparrow\ \frac{a = \mathit{store}(b, \bar{i}, v) \qquad w = b'[\bar{j}] \qquad b \sim b'}{\bar{i} = \bar{j} \;\;\Big|\;\; w = a[\bar{j}]}$$

$$\mathsf{map}\Downarrow\ \frac{a = \mathit{map}_f(b_1, \ldots, b_m) \qquad w = a'[\bar{i}] \qquad a \sim a'}{w = f(b_1[\bar{i}], \ldots, b_m[\bar{i}])}$$

$$\mathsf{map}\Uparrow\ \frac{a = \mathit{map}_f(b_1, \ldots, b_m) \qquad w = b'[\bar{i}] \qquad b' \sim b_k \text{ for some } k \in \{1, \ldots, m\}}{a[\bar{i}] = f(b_1[\bar{i}], \ldots, b_{k-1}[\bar{i}], w, b_{k+1}[\bar{i}], \ldots, b_m[\bar{i}])}$$

$$\mathsf{proj}\Downarrow\ \frac{a = \mathit{proj}_k(b, j) \qquad w = a'[\bar{i}] \qquad a \sim a'}{w = b[i_1, \ldots, i_{k-1}, j, i_k, \ldots, i_{n-1}]}$$

$$\mathsf{proj}\Uparrow\ \frac{a = \mathit{proj}_k(b, j) \qquad w = b'[\bar{i}] \qquad b \sim b'}{j \neq i_k \;\;\Big|\;\; w = a[i_1, i_2, \ldots, i_{k-1}, i_{k+1}, \ldots, i_n]}$$

$$\mathsf{arrayStore}\Downarrow\ \frac{a = \mathit{arrayStore}_k(b, j, c) \qquad w = a'[\bar{i}] \qquad a \sim a'}{j = i_k \wedge w = c[i_1, \ldots, i_{k-1}, i_{k+1}, \ldots] \;\;\Big|\;\; j \neq i_k \wedge w = b[\bar{i}]}$$

$$\mathsf{arrayStore}\Uparrow_1\ \frac{a = \mathit{arrayStore}_k(b, j, c) \qquad w = b'[\bar{i}] \qquad b \sim b'}{j = i_k \;\;\Big|\;\; w = a[\bar{i}]}$$

$$\mathsf{arrayStore}\Uparrow_2\ \frac{a = \mathit{arrayStore}_k(b, j, c) \qquad w = c'[\bar{i}] \qquad c \sim c'}{w = a[i_1, \ldots, i_{k-1}, j, i_k, \ldots, i_{n-1}]}$$

$$\mathsf{ext}\ \frac{a : (\sigma^n \Rightarrow \tau) \qquad b : (\sigma^n \Rightarrow \tau)}{a = b \;\;\Big|\;\; \exists \bar{i} : \sigma^n.\ a[\bar{i}] \neq b[\bar{i}]} \qquad \mathsf{freshIdx}\ \frac{i_1, \ldots, i_k : \sigma}{\exists j : \sigma.\ j \neq i_1 \wedge \cdots \wedge j \neq i_k}$$

$$\mathsf{read}\ \frac{a : (\sigma^n \Rightarrow \tau) \qquad \bar{i} : \sigma^n}{\exists v : \tau.\ v = a[\bar{i}]} \qquad \mathsf{readCong}\ \frac{v = a[\bar{i}] \qquad w = b[\bar{j}] \qquad a \sim b}{\bar{i} \neq \bar{j} \;\;\Big|\;\; \bar{i} = \bar{j} \wedge v = w}$$

where $a, b, c, d, k \in \mathbb{Z}$. A complex number is then represented by a five-tuple (a, b, c, d, k). Although the considered set of numbers is only a small subset of $\mathbb{C}$, it is closed under the operations needed to encode quantum gates, and it can arbitrarily closely approximate any complex number. For this, note that $(a, 0, c, 0, k)$ represents $\frac{1}{\sqrt{2}^k}(a + c\omega^2) = \frac{a}{\sqrt{2}^k} + \frac{ci}{\sqrt{2}^k}$, and pick suitable a, c, and k. The representation is also sufficient to describe a set of quantum gates that can implement universal quantum computation (Table 3).

4 A Decision Procedure for Cartesian Arrays

We now present a decision procedure for quantifier-free CaAL. Our calculus is an extension of the calculus for CAL [11] with rules for the *proj* and *arrayStore* operations. For the sake of presentation, we use the setting of analytic tableaux [14], although the same proof rules can be used also in a model-constructing calculus [11].

As a simplifying assumption, in this section we furthermore require that the index sorts σ of an array sort $(\sigma^n \Rightarrow \tau)$ represent *infinite* domains. This assumption can be lifted in the same way as for CAL [11], but the details are orthogonal to the task of supporting the new array operations.

4.1 Preliminaries

A tableau [14] is a finite tree growing downwards, in which each node is labelled with a formula, the root is labelled with the formula to be refuted, and the children of each node are derived from the formulas on the branch leading to the node using one of the available proof rules. We assume a tableau calculus equipped with a set of standard rules [14]: (i) α- and β-rules for eliminating Boolean connectives $\wedge, \vee$; (ii) δ-rules for eliminating existential quantifiers $\exists$; (iii) rules for reasoning about positive and negative equalities $x = y$ between variables, which include rules for closing proof branches; (iv) rules implementing a decision procedure for the base theory T_B.

Our calculus operates on *flat* formulas, which are formulas in which functions f only occur in equations $y = f(\bar{x})$ in positive positions, i.e., underneath an even number of negations, with $y, \bar{x}$ being variables. Every formula can be converted to a flat formula by introducing a linear number of new variables.

We define proof rules using the following notation:

$$\textsf{rule}\ \frac{\phi_1 \qquad \phi_2 \qquad \cdots \qquad \phi_k}{\psi_1 \;\Big|\; \cdots \;\Big|\; \psi_m}$$

The rule is applicable if the premises $\phi_1, \ldots, \phi_k$ occur on a proof branch, and has the effect of expanding the tableau: the proof branch is split into m new branches, to which the formulas $\psi_1, \ldots, \psi_m$, respectively, are appended.

In the premises of a rule, we frequently include assumptions $x \sim y$ that require that the equality $x = y$ follows from positive equalities between variables on the proof branch. We also use premises $x : \sigma$, stating that x is a variable of sort σ occurring on the proof branch.

4.2 Proof Rules

The rules of our calculus are shown in Table 4. The rules idx, $\mathsf{K}\Downarrow$, $\mathsf{store}\Downarrow$, $\mathsf{store}\Uparrow$, $\mathsf{map}\Downarrow$, $\mathsf{map}\Uparrow$ coincide with the rules used for CAL [11], and define the semantics of the operators K, *store*, and *map*. Extensionality is implemented by the rule ext, which can be applied for any two array variables a, b of the same type occurring on a branch.

The semantics of *proj* and *arrayStore* is defined, in a similar way as for *store*, by upward and downward propagation of array reads. Since $\mathit{arrayStore}_k(b, j, c)$ combines two arrays b, c into a single new array, downward propagation has to route reads either to b or to c. Upward propagation from c is always possible, while reads on b can only be propagated if they are not overwritten by c.

For sake of presentation, we write the conclusion in the rules $\mathsf{map}\Downarrow$, $\mathsf{map}\Uparrow$, and ext in non-flat form, and assume that the transformation to a flat formula happens implicitly by adding existentially quantified variables representing the sub-terms.

Congruence reasoning is necessary only for array reads, and implemented using the rule readConq. For simplicity, in our formulation the rule splits over the cases $\bar{i} \neq \bar{j}$ and $\bar{i} = \bar{j}$, and effectively searches for an arrangement of the index variables satisfying a formula. An actual implementation could rely on equality propagation being performed by a theory combination procedure.

As one of the more tricky points, the completeness of the calculus sometimes requires new array reads to be generated. This aspect is covered by the rules $\epsilon_{\not\approx}$ and $\epsilon\delta$ in CAL [11], which are rules that can, however, not directly be used in our setting of multi-dimensional arrays. To obtain completeness, our calculus sometimes has to construct reads by combining different index variables occurring on a branch, and sometimes invent index values that are distinct from all indexes occurring in a formula. The introduction of corresponding new reads is handled by the rules freshIdx and read.

Example 3. Consider arrays $a, b : (\mathbb{Z}^2 \Rightarrow \mathbb{Z})$, and the formulas

$$\mathit{proj}_1(a, i) = K(42) \wedge \mathit{proj}_2(a, j) = K(43) \tag{10}$$

$$a = K(42) \wedge b = \mathit{store}(a, (i, i), 43) \wedge \mathit{proj}_1(b, i) = K(43) \tag{11}$$

Both formulas are unsatisfiable, but cannot be refuted using the rules discussed so far. In (10), no reads $a[\cdots]$ exist, so that no propagations can be performed by any of the rules. It is necessary to identify the constraints on the value $a[i, j]$ as contradictory. The rule read can be used to introduce a new formula $\exists v.\ v = a[i, j]$ on a proof branch, after which the rules $\mathit{proj}\Uparrow$ and $\mathsf{K}\Downarrow$ can be applied.

To show that (11) is unsatisfiable, we need to consider a point (i, j) with $j \neq i$ and derive that $a[i, j] = b[i, j] = 42$, and contradicting $\mathit{proj}_1(b, i) = K(43)$. The introduction of a fresh index value j (different from i) is handled by the rule freshIdx, which relies on the index sort σ representing an infinite domain. Once the existence of an index $j \neq i$ has been asserted, the rule read can be used to introduce an equation $v = a[i, j]$, and the contraction be derived.

4.3 Correctness and Complexity

Theorem 3. *The presented tableau calculus is sound and complete for flat quantifier-free CaAL formulas: there is a closed tableau for a formula ϕ if and only if ϕ is unsatisfiable.*

Proof. Soundness: As usual, we identify each proof branch with the conjunction of its formulas and a tableau with the disjunction of its proof branches. It can be shown that the tableau before expansion using a proof rule is equi-satisfiable to the tableau before the expansion, modulo the array axioms in Table 2.

Completeness: We make the simplifying assumption that ϕ only contains arrays with (infinite) index sort σ and value sort τ, and in particular that array sorts are not nested. Completeness for the general case follows by recursively applying model construction.

Consider then the systematic construction of a tableau for a formula ϕ by exhaustively applying proof rules under the following restrictions: (i) regularity, i.e., rules are only applied if they lead to new formulas being added to each generated branch; (ii) rule freshIdx can only be applied once on a branch, only after ext has been applied to all pairs a, b of array variables on the branch, and choosing $i_1, \ldots, i_k$ as the set of all variables of sort σ on the branch.

Observe that this systematic application of rules terminates: the calculus never introduces new array variables so that only finitely many applications of ext are possible. Note that ext and freshIdx are the only rules introducing new index variables. Since freshIdx is applied at most once on a branch, the set of index variables is bounded, and there is only a bounded number of array reads $v = a[\bar{i}]$.

Assume now that a tableau for ϕ cannot be closed, i.e., has at least one branch B that cannot be closed, although all possible rule applications have been performed. We extract a model of ϕ from B. Suppose that $M_T = (D_T, I_T)$ is a model that interprets the non-array-variables (including index variables), satisfying all literals on B that do not contain array variables, and denote the equivalence class of an array variable a on B by $[a] = \{b \mid a \sim b\}$. Extending I_T, we construct an interpretation I with $I((\sigma^n \Rightarrow \tau)) = I_T(\sigma)^n \to I_T(\tau)$ being a function space, and the theory functions $\cdot[\cdot], \mathit{store}, K, \mathit{map}_f, \mathit{proj}$ and $\mathit{arrayStore}$ having their expected meaning. I is constructed in such a way that all array literals on B are satisfied; the satisfaction of compound formulas on B, and in particular of ϕ, then follows like in the standard Hintikka construction [14].

The interpretation $I(a)$ of an array variable $a : (\sigma^n \Rightarrow \tau)$ is derived from the array reads on $[a]$ occurring on B. The main difficulty is to consistently interpret the (infinitely many) elements of the array that are not mentioned explicitly on B. For this, denote the index variable introduced by the unique freshIdx application on B by ϵ, and observe that its value $I_T(\epsilon)$ is distinct from the value of all other index variables. We will use values read from $I_T(\epsilon)$-locations as default values for the arrays. Let

$$R_a = \{(\langle I_T(i_1), \ldots, I_T(i_n)\rangle, I_T(v)) \mid v = b[\bar{i}] \text{ occurs on } B \text{ and } a \sim b\}$$

be the set of array reads for $a : (\sigma^n \Rightarrow \tau)$. The relation R_a describes a non-empty, consistent (but partial) valuation of the array elements, due to the exhaustive application of rules read and readCong.

The gaps in R_a will be filled with default values introduced by ϵ. For this, we define a precedence ordering $\preceq \subseteq I_T(\sigma)^* \times I_T(\sigma)^*$ over index vectors; intuitively,

$\bar{c} \preceq \bar{d}$ if $\bar{c}$ and $\bar{d}$ agree in all components, unless $d_k = I_T(\epsilon)$, which is interpreted as don't-care:

$$\langle c_1, \ldots, c_k \rangle \preceq \langle d_1, \ldots, d_m \rangle \text{ iff } k = m \text{ and } \forall i \in \{1, \ldots, k\} : \ c_i = d_i \vee d_i = I_T(\epsilon)$$

The value of array variable $I(a) \in I((\sigma^n \Rightarrow \tau))$ is then:

$$I(a) = \left\{ (\bar{c}, x) \mid \begin{array}{l} (\bar{d}, x) \in R_a, \text{ where } \bar{c} \preceq \bar{d} \\ \text{and for all } (\bar{d}', x') \in R_a : \text{ if } \bar{c} \preceq \bar{d}' \text{ then } \bar{d} \preceq \bar{d}' \end{array} \right\}$$

To see that $I(a)$ is functionally consistent, note that whenever $(\bar{d}, x)$ and $(\bar{d}', x')$ exist in R_a such that $\bar{c} \preceq \bar{d}$ and $\bar{c} \preceq \bar{d}'$, then there is also some $(\bar{d}'', x'') \in R_a$ such that $\bar{c} \preceq \bar{d}'' \preceq \bar{d}, \bar{d}'$. This is because the rule read has been applied exhaustively.

It remains to be shown that I satisfies all array literals. By construction, equations $a = b$ will be satisfied. To see that equations $v = a[\bar{i}]$ hold, note that $I(a) \supseteq R_a$. Equations $a \neq b$ are satisfied due to the exhaustive application of ext: there has to be some vector $\bar{i}$ of index variables such that $a[\bar{i}] \neq b[\bar{i}]$.

All other array literals are positive equations of the form $x = f(\bar{y})$, and hold because exhaustive propagation of read atoms was performed. As an example, consider an equation $a = \mathit{proj}_k(b, j)$; it has to be shown that $I(a) = \{(\langle c_1, \ldots, c_{k-1}, c_{k+1}, \ldots, c_n \rangle, x) \mid (\bar{c}, x) \in I(b), c_k = I_T(j)\}$. Observe that $R_a = \{(\langle c_1, \ldots, c_{k-1}, c_{k+1}, \ldots, c_n \rangle, x) \mid (\bar{c}, x) \in R_b, c_k = I_T(j)\}$ due to the rules $\mathit{proj}\Downarrow$ and $\mathit{proj}\Uparrow$. Consider then a point $(\bar{c}, x) \in I(a)$, defined by $(\bar{d}, x) \in R_a$, and the corresponding index vectors $\bar{c}' = \langle c_1, \ldots, c_{k-1}, I_T(j), c_k, \ldots, c_{n-1} \rangle$ and $\bar{d}' = \langle d_1, \ldots, d_{k-1}, I_T(j), d_k, \ldots, d_{n-1} \rangle$ in R_b, and show that $(\bar{c}', x) \in I(b)$ is defined by $(\bar{d}', x) \in R_b$. □

The proof of the theorem highlights the restrictions necessary to obtain a decision procedure for CaAL: all rules should be applied under the condition of regularity, and the rule freshIdx has to be restricted to at most one application per branch, and only after applications of ext have been performed.

To evaluate runtime, like in the proof of Theorem 3 we make the assumption that there are no nested array sorts, i.e., index and value sorts are themselves not arrays. To avoid degenerate cases when evaluating runtime, we assume that a formula ϕ cannot be smaller than the maximum arity of occurring array variables. We then get:

Lemma 1. *The satisfiability problem of quantifier-free CaAL formulas ϕ without nested array sorts is in NEXPTIME, assuming that the satisfiability problem of the base theory is in NP.*

Proof. This follows from the proof of Theorem 3. On every branch, the rule ext can be applied at most quadratically often, and the number of index variables occurring on a branch is polynomial in the size of the input formula ϕ. The number of distinct read atoms $v = a[\bar{i}]$ that can be introduced on a branch, and therefore the number of rule applications altogether is then polynomially bounded by the number of variables in ϕ, and exponentially bounded in the maximum arity of array variables in ϕ. After exhaustive application of the rules in Table 4, solving an at most exponential number of base theory formulas (with at most exponential size) on a branch is in NEXPTIME. □

4.4 Optimizations

The calculus and decision procedure are primarily designed with simplicity in mind, rather than focusing on practical efficiency. Although the procedure's complexity may not be reduced below NEXPTIME, incorporating various optimizations can yield significant practical improvements. Two obvious improvements to be considered are: (i) The detection of **linear array variables**, which are essentially variables that are assigned to at most once in array literals [11]. It is enough to perform upward propagation (rules $\Uparrow$) only for non-linear variables. (ii) The **restriction of the number of reads** introduced using the rule read. In practice, only a few of the generated equations are actually needed to ensure completeness. Instead of generating all possible reads eagerly, a procedure could focus on the other rules first, and only introduce additional reads when it is detected that default values are missing for some sub-arrays. We believe that other refinements presented in [11] can be carried over to our decision procedure as well.

Table 5. Experimental results. We list the **circuit** name, the number of **qubits** and **gates** in the circuit, the verification **result**, and the execution **time**.

circuit	qubits	gates	result	time	circuit	qubits	gates	result	time
H^2	1	2	OK	3.1 s	H^2 (bug)	1	2	bug	3.0 s
BV	1	3	OK	3.2 s	BV (bug)	1	3	bug	3.3 s
BV	2	5	OK	6.4 s	BV	5	13	OK	1 m 59.0 s
BV	3	8	OK	16.8 s	BV	6	15	OK	9 m 13 s
BV	4	10	OK	43.2 s	BV	7	18	OK	50 m 54 s
$\text{Grover}_{\text{Single-Comp}}$	2	17	OK	5.2 s	$\text{Grover}_{\text{Single-Comp}}$	4	85	OK	51.7 s
$\text{Grover}_{\text{All-Comp}}$	2	17	OK	6.8 s	$\text{Grover}_{\text{All-Comp}}$	4	85	OK	3 m 53 s
$\text{Grover}_{\text{Single-Iter}}$	1	9	OK	3.2 s	$\text{Grover}_{\text{All-Iter}}$	1	9	OK	3.8 s
$\text{Grover}_{\text{Single-Iter}}$	2	15	OK	4.9 s	$\text{Grover}_{\text{All-Iter}}$	2	15	OK	14.2 s
$\text{Grover}_{\text{Single-Iter}}$	3	21	OK	8.4 s	$\text{Grover}_{\text{All-Iter}}$	3	21	OK	37.9 s
$\text{Grover}_{\text{Single-Iter}}$	4	27	OK	17.1 s	$\text{Grover}_{\text{All-Iter}}$	4	27	OK	4 m 51 s
$\text{Grover}_{\text{Single-Iter}}$	5	33	OK	46.9 s	$\text{Grover}_{\text{All-Iter}}$	5	33	OK	57 m 2 s

5 Preliminary Experimental Result

We have implemented the decision procedure proposed for CaAL, the encoding of quantum gates using array operations, and of complex numbers as five-tuples of integers in the SMT solver Princess [26]. The implementation is still a proof of concept and largely unoptimized, so that the results reported in this section should be considered preliminary. We evaluate the performance of CaAL based on a set of benchmarks for quantum circuit verification. All experiments were conducted on a server with an AMD EPYC 7742 64-core processor (1.5 GHz), 1,152 GiB of RAM, and a 1 TB SSD running Ubuntu 20.04.5 LTS but were run with only one core for the sake of fairness. Files to reproduce the experiment can be found in https://zenodo.org/record/7970588. The experimental results are shown in Table 5. Specifically, we tested four different verification problems with different circuit sizes.

- H^2: Two consecutive H gates equal to identity.
- BV: The (complex) amplitudes of the output quantum state from a Bernstein-Vazirani's [4] circuit have no imaginary parts.
- $\text{Grover}_{\text{XXX-Comp}}$: The Grover's [17] circuit has a probability of 90% to find the correct answer.
- $\text{Grover}_{\text{XXX-Iter}}$: Each Grover iteration [17] increases the possibility of finding the correct answer.

For Grover's algorithm, XXX = Single means we check the correctness of the circuit against a specific oracle, and XXX = All means we check against all possible oracles. We manually injected two bugs (by altering one gate) into two examples to demonstrate bug-catching capability. With a timeout of 60min, our implementation can analyze circuits with at most 7 qubits and at most 85 gates, which are still relatively small circuits. Analyzing the results, we discovered that, in particular, the H gates used to create a superposition state at the beginning of a circuit are challenging for the array decision procedure, as they lead to an exponential number of array reads being created.

6 Conclusions

We have presented CaAL, an expressive logic of extensional arrays, with operations for reading and storing values, creating constant arrays, a point-wise extension of functions on array values to arrays, projection of arrays, and updating array slices. We have established that checking the satisfiability of quantifier-free CaAL formulas is NEXPTIME-complete, for a base theory in NP and non-nested arrays. The root cause for the complexity of CaAL (as opposed to the NP complexity of CAL and the standard theory of arrays) is that formulas can be constructed in which a cell in one array has dependencies to an exponential number of cells in another array. In our decision procedure, such situations lead to an exponential number of reads generated during propagation. High degrees of dependency are typical, however, for quantum circuits.

We believe that CaAL is a suitable framework for reasoning about quantum circuits. Due to the expressiveness of the logic, the encoding of quantum gates becomes remarkably succinct and elegant (Table 3), and easily understandable both for researchers in quantum circuit verification and people in automated reasoning. While theoretically optimal, we consider the decision procedure proposed for CaAL only as a first step: the high complexity of CaAL implies that brute-force approaches like saturation are unlikely to scale to interesting instances. As future work, we therefore plan to explore the use of abstraction methods and of more succinct array representations in the decision procedure, thus making it possible to exploit the highly structured nature of typical quantum circuits in the solving process. We also plan to investigate whether interesting fragments of CaAL with lower complexity can be identified.

Acknowledgements. This work has been partially funded by the Swedish Research Council (VR) under grant 2018-04727, the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), the Wallenberg project UPDATE, and the NSTC QC project under Grant no. NSTC 111-2119-M-001-004- and 112-2119-M-001-006-.

References

1. Amy, M.: Towards large-scale functional verification of universal quantum circuits. In: Quantum Physics and Logic (2018)
2. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
3. Bauer-Marquart, F., Leue, S., Schilling, C.: symqv: automated symbolic verification of quantum programs. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) FM 2023. LNCS, vol. 14000, pp. 181–198. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-27481-7_12
4. Bernstein, E., Vazirani, U.V.: Quantum complexity theory. In: Kosaraju, S.R., Johnson, D.S., Aggarwal, A. (eds.) Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16–18, 1993, San Diego, CA, USA, pp. 11–20. ACM (1993). https://doi.org/10.1145/167088.167097
5. Bonacina, M.P., Graham-Lengrand, S., Shankar, N.: CDSAT for nondisjoint theories with shared predicates: arrays with abstract length. In: Déharbe, D., Hyvärinen, A.E.J. (eds.) Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories Co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) Part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, 11–12 August 2022. CEUR Workshop Proceedings, vol. 3185, pp. 18–37. CEUR-WS.org (2022). https://ceur-ws.org/Vol-3185/paper9712.pdf
6. Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. J. Satisf. Boolean Model. Comput. **6**(1–3), 165–201 (2009). https://doi.org/10.3233/sat190067
7. Chen, Y., Chung, K., Lengál, O., Lin, J., Tsai, W., Yen, D.: An automata-based framework for verification and bug hunting in quantum circuits (2023). https://doi.org/10.48550/arxiv.2301.07747, https://arxiv.org/abs/2301.07747. To appear at PLDI 2023
8. Christ, J., Hoenicke, J.: Weakly equivalent arrays. In: Lutz, C., Ranise, S. (eds.) FroCoS 2015. LNCS (LNAI), vol. 9322, pp. 119–134. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24246-0_8
9. Coecke, B., Duncan, R.: Interacting quantum observables: categorical algebra and diagrammatics. New J. Phys. **13**(4), 043016 (2011). https://doi.org/10.1088/1367-2630/13/4/043016
10. Dawson, C.M., Nielsen, M.A.: The Solovay-Kitaev algorithm. arXiv preprint quant-ph/0505030 (2005)
11. De Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: 2009 Formal Methods in Computer-Aided Design, pp. 45–52. IEEE (2009)

12. Falke, S., Merz, F., Sinz, C.: Extending the theory of arrays: memset, memcpy, and beyond. In: Cohen, E., Rybalchenko, A. (eds.) VSTTE 2013. LNCS, vol. 8164, pp. 108–128. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54108-7_6
13. Feng, Y., Yu, N., Ying, M.: Model checking quantum Markov chains. J. Comput. Syst. Sci. **79**(7), 1181–1198 (2013). https://doi.org/10.1016/j.jcss.2013.04.002
14. Fitting, M.C.: First-Order Logic and Automated Theorem Proving, 2nd edn. Springer, New York (1996). https://doi.org/10.1007/978-1-4612-2360-3
15. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_52
16. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_52
17. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Miller, G.L. (ed.) Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, 22–24 May 1996, pp. 212–219. ACM (1996). https://doi.org/10.1145/237814.237866
18. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press, Cambridge (2009)
19. Hietala, K., Rand, R., Hung, S.H., Wu, X., Hicks, M.: Verified optimization in a quantum intermediate representation. arXiv preprint arXiv:1904.06319 (2019)
20. Hoenicke, J., Schindler, T.: Solving and interpolating constant arrays based on weak equivalences. In: Enea, C., Piskac, R. (eds.) VMCAI 2019. LNCS, vol. 11388, pp. 297–317. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11245-5_14
21. Mateus, P., Ramos, J., Sernadas, A., Sernadas, C.: Temporal Logics for Reasoning about Quantum Systems, pp. 389–413. Cambridge University Press, Cambridge (2009). https://doi.org/10.1017/CBO9781139193313.011
22. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information: 10th Anniversary Edition, 10th edn. Cambridge University Press, USA (2011)
23. Papadimitriou, C.H., Yannakakis, M.: A note on succinct representations of graphs. Inf. Control **71**(3), 181–185 (1986)
24. Perdrix, S.: Quantum entanglement analysis based on abstract interpretation. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 270–282. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69166-2_18
25. Raya, R., Kunčak, V.: NP satisfiability for arrays as powers. In: Finkbeiner, B., Wies, T. (eds.) VMCAI 2022. LNCS, vol. 13182, pp. 301–318. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-94583-1_15
26. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 274–289. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89439-1_20
27. Sheng, Y., et al.: Reasoning about vectors using an SMT theory of sequences. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 125–143. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_9
28. Sipser, M.: Introduction to the theory of computation. ACM SIGACT News **27**(1), 27–29 (1996)
29. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.: A decision procedure for an extensional theory of arrays. In: Proceedings 16th Annual IEEE Symposium on Logic in Computer Science, pp. 29–37. IEEE (2001)

30. Tsai, Y., Jiang, J.R., Jhang, C.: Bit-slicing the Hilbert space: scaling up accurate quantum circuit simulation. In: 58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, 5–9 December 2021, pp. 439–444. IEEE (2021). https://doi.org/10.1109/DAC18074.2021.9586191
31. Wang, Q., Appel, A.W.: A solver for arrays with concatenation. J. Autom. Reason. **67**(1), 4 (2023). https://doi.org/10.1007/s10817-022-09654-y
32. Xu, M., Fu, J., Mei, J., Deng, Y.: Model checking QCTL plus on quantum Markov chains. Theor. Comput. Sci. **913**, 43–72 (2022). https://doi.org/10.1016/j.tcs.2022.01.044
33. Xu, M., et al.: Quartz: superoptimization of quantum circuits. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 625–640 (2022)
34. Yu, N., Palsberg, J.: Quantum abstract interpretation. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 542–558 (2021)
35. Zulehner, A., Wille, R.: Advanced simulation of quantum computations. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **38**(5), 848–859 (2019). https://doi.org/10.1109/TCAD.2018.2834427

SAT-Based Subsumption Resolution

Robin Coutelier[1](✉), Laura Kovács[2], Michael Rawson[2], and Jakob Rath[2]

[1] U. Liège, Liège, Belgium
robin.coutelier@student.uliege.be
[2] TU Wien, Vienna, Austria

Abstract. Subsumption resolution is an expensive but highly effective simplifying inference for first-order saturation theorem provers. We present a new SAT-based reasoning technique for subsumption resolution, without requiring radical changes to the underlying saturation algorithm. We implemented our work in the theorem prover Vampire, and show that it is noticeably faster than the state of the art.

1 Introduction

Saturation-based proof search is a popular approach to first-order theorem proving [6,14,18]. In addition to efficient inference systems [1,8], saturation provers also implement *redundancy elimination* to reduce the size of the search space. Redundancy elimination deletes clauses from the search space by showing them to be logical consequences of other (smaller) clauses, and therefore redundant. However, checking whether a first-order formula is implied by another first-order formula is undecidable, and so eliminating redundant clauses is in general undecidable too. In practice, saturation systems apply cheaper conditions for redundancy elimination, such as removing equational tautologies by congruence closure or deleting subsumed clauses by establishing multiset inclusion. Recently, SAT solving has been applied to efficiently detect and remove subsumed clauses [10]. *We extend SAT-based reasoning in first-order theorem proving to a combination of subsumption and resolution,* ***subsumption resolution*** [2] *(Sect. 4).*

Both subsumption and subsumption resolution are NP-complete [4]. To improve efficiency in practice, we (i) encode subsumption resolution as SAT formulas over (match) set constraints (Sect. 5) and (ii) directly integrate CDCL SAT solving for checking subsumption resolution in first-order theorem proving (Sect. 6). We implement our approach in the theorem prover Vampire [6], improving the state-of-the-art in first-order reasoning (Sect. 7).

Related Work. Subsumption and subsumption resolution are some of the most powerful and frequently used redundancy criteria in saturation-based provers. Subsumption resolution is supported as *contextual literal cutting* in [14], along with efficient approaches for detecting multiset inclusions among clauses [6,13,18]. Special cases of unit deletion as a by-product of subsumption tests are also proposed in [16]. Much attention has been given to refinements of *term indexing*

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 190–206, 2023.
https://doi.org/10.1007/978-3-031-38499-8_11

[13,16] to drastically reduce the set of candidate clauses checked for subsumption. Recently, these approaches have been complemented by SAT solving [10], reducing subsumption checking to SAT. Our work generalises this approach by solving for both subsumption and subsumption resolution via SAT.

SAT solvers have been applied widely to first-order theorem proving, including but not limited to AVATAR [17], instance-based methods [5], heuristic grounding [14], global subsumption [12] and combinations thereof [11], but using SAT solvers for classical subsumption methods is under-explored. To the best of our knowledge, SAT solving for subsumption resolution has so far not been addressed in the landscape of automated reasoning.

2 Illustrative Examples and Main Contributions

Let us illustrate a few challenges of subsumption resolution, which motivate our approach to solving it (Sect. 4). Given a pair of clauses L and M, denoted as (L, M), the problem is to decide whether M can be simplified by L via a special case of logical consequence. In Fig. 1 we show examples where it is not obvious for which pairs (L_i, M_i) subsumption resolution can be applied.

$L_1 := p(x_1, x_2) \vee p(f(x_2), x_3)$ $M_1 := p(g(y_1), c) \vee \neg p(f(c), e)$	$L_2 := p(x_1) \vee q(x_2)$ $M_2 := \neg p(y) \vee \neg q(c)$
$L_3 := p(x_1) \vee q(x_1, x_2) \vee \neg p(x_2)$ $M_3 := \neg p(y) \vee q(y, y)$	$L_4 := p(x_1) \vee q(x_2) \vee r(x_3)$ $M_4 := \neg p(y_1) \vee q(c)$

Fig. 1. Illustrative examples.

In fact, subsumption resolution can only be applied to (L_1, M_1). Later, we show how our approach determines that M_1 can be shortened in the presence of L_1 (Example 3.1), but also how the remaining pairs cannot apply subsumption resolution (Examples 5.1, 5.2, and 4.1). For example, (L_4, M_4) is filtered by *pruning* to bypass the SAT routine altogether.

Our Contributions

1. We cast the problem of subsumption resolution over pairs of first-order formulas (L, M) as a SAT problem (Theorem 5.1), ensuring any instance of subsumption resolution is a model of this SAT problem.
2. We tailor encodings of subsumption resolution (Sects. 5.1–5.2) for effective SAT-based subsumption resolution (Algorithm 1).
3. We integrate our approach into the saturation loop, solving for subsumption and subsumption resolution simultaneously (Sect. 6).
4. We implement our work in the theorem prover Vampire and showcase our practical gains in first-order proving (Sect. 7).

3 Preliminaries

We assume familiarity with first-order logic with equality. We include standard Boolean connectives and quantifiers in the language, and the constants $\top, \bot$ for truth and falsehood. We use x, y, z for first-order variables, c, d, e for constants, f, g for functions, p, q, r for atoms, l, m for literals, and L, M for clauses, all potentially with indices. If L is a clause $l_1 \vee \ldots \vee l_n$, we sometimes consider it as a multiset of its literals l_i, and write $|L|$ for its cardinality (i.e. the number n of literals in L). The empty clause is denoted $\square$. Free variables are universally quantified. An expression E is a term, atom, literal, clause, or formula.

Substitutions and Matches. A substitution σ is a (partial) mapping from variables to terms. The result of applying a substitution σ to an expression E is denoted $\sigma(E)$ and is the expression obtained by simultaneously replacing each variable x in E by $\sigma(x)$. For example, the application of $\sigma := \{x \mapsto f(c)\}$ to the clause $L := \{p(x), q(x, y)\}$ yields $\sigma(L) = \{p(f(c)), q(f(c), y)\}$. Note that $\sigma(L)$ is a logical consequence of L.

A *matching substitution*, in short a *match*, between literals l and m is a substitution σ such that $\sigma(l) = m$. For example, the match of $p(x)$ onto $p(f(c))$ is $\{x \mapsto f(c)\}$. Two matches are *compatible* and can be combined in the same substitution iff they do not assign different terms to the same variable. For example, the substitutions $\{x \mapsto f(c), y \mapsto g(d)\}$ and $\{x \mapsto f(c), z \mapsto h(e)\}$ are compatible, but $\{x \mapsto f(c)\}$ and $\{x \mapsto g(c)\}$ are not.

Saturation and Redundancy. Many first-order systems apply the superposition calculus [1] in a saturation loop [8]. Given an input set F of clauses, saturation iteratively derives logical consequences and adds them to F. By soundness and completeness of superposition, if $\square$ is derived the system can report unsatisfiability of F; if $\square$ is not encountered and no further clauses can be derived, the system reports satisfiability of F.

Saturation is more efficient when F is as small as possible. For this reason, saturation-based provers also employ *simplifying* inferences. Simplifying inferences reduce the number or size of clauses in F. This is formalised using the following notion of *redundancy*: a ground clause M is redundant in a set of ground clauses F if M is a logical consequence of clauses in F that are strictly smaller than M w.r.t. a fixed simplification ordering $\succ$. A non-ground clause M is redundant in a set of clauses F if each ground instance of M is redundant in the set of ground instances of F. If M is redundant in F, then M can be removed from F while retaining completeness.

Subsumption. A clause L *subsumes* a distinct clause M iff there is a substitution σ such that

$$\sigma(L) \subseteq_M M \tag{1}$$

where $\subseteq_M$ denotes multiset inclusion. We also say that M is *subsumed* by L. Note that subsumed clauses are redundant.

Removing subsumed clauses M from the search space F is implemented through a simplifying rule, checking condition (1) over pairs of clauses (L, M) from F. Matches between every literal in L to some literal in M are checked; if a compatible set of matches is found, then M can be removed from F.

Subsumption Resolution. Subsumption resolution aims to remove one redundant literal from a clause. Clauses M and L are said to be the main and side premise of subsumption resolution, respectively, iff there is a substitution σ, a set of literals $L' \subseteq L$ and a literal $m' \in M$ such that

$$\sigma(L') = \{\neg m'\} \quad \text{and} \quad \sigma(L \setminus L') \subseteq M \setminus \{m'\}. \tag{2}$$

If so, M can be replaced by $M \setminus \{m'\}$. Subsumption resolution is hence the rule

$$(\texttt{SR}) \qquad \frac{L \qquad \cancel{M}}{M \setminus \{m'\}}$$

We indicate the deletion of a clause M by drawing a line through it ($\cancel{M}$), and we refer to the literal m' of M as the *resolution literal* of SR. Intuitively, subsumption resolution is binary resolution followed by subsumption of one of its premises by the conclusion. However, by combining two inferences into one it can be treated as a simplifying inference, which is advantageous from the perspective of proof search dynamics.

Example 3.1. Consider L_1, M_1 of Fig. 1. Subsumption resolution is applied by using the substitution $\sigma \coloneqq \{x_1 \mapsto g(y_1), x_2 \mapsto c, x_3 \mapsto e\}$. Note that $\sigma(L_1) = p(g(y_1), c) \vee p(f(c), e)$. $\sigma(L_1)$ and M_1 can be resolved to obtain $p(g(y_1), c)$. The clause $p(g(y_1), c)$ subsumes M_1, since it is a sub-multiset of M_1. We have

$$\frac{p(x_1, x_2) \vee p(f(x_2), x_3) \qquad \cancel{p(g(y_1), c) \vee \neg p(f(c), e)}}{p(g(y_1), c)}$$

4 SAT-Based Subsumption Resolution

We describe the main steps of our SAT-based approach for deciding the applicability of subsumption resolution on a pair (L, M) of clauses. The core of our work solves (2) by finding match substitutions between literals in L and M. Our technique is summarised in Algorithm 1.

Pruning. The first step of Algorithm 1 *prunes* pairs (L, M) of clauses that cannot be simplified by subsumption resolution due to a syntactic restriction over symbols in L and M, *viz.* whether the set of predicates in L is a subset of the predicates in M. If not, then there is a literal in L that cannot be matched to any literal in M, and hence subsumption resolution cannot be applied.

Example 4.1. The clause pair (L_4, M_4) from Fig. 1 is pruned by Algorithm 1: the set of predicates in L_4 and M_4 are respectively $\{p, q, r\}$ and $\{p, q\}$, implying that the literal $r(x_3)$ of L_4 cannot be matched to any literal in M_4.

Algorithm 1. SAT-based subsumption resolution over pair (L, M) of clauses

```
$ms \leftarrow$ createMatchSet()
$solver \leftarrow$ createSatSolver($ms$)
procedure SubsumptionResolution($L, M$)
    if pruned($L, M$) then
        return NoSubsumptionResolution
    if fillMatchSet($ms, L, M$) is false then
        return NoSubsumptionResolution
    encodeConstraints($solver$, $ms$)
    if $solver$.solve() is SAT then
        return buildConclusion($solver$.getSolution(), $M$)    ▷ conclusion of subsumption resolution
    return NoSubsumptionResolution
```

Match Set. The *match set* of Algorithm 1 computes matching substitutions over literals of L and M. The match set ms consists of a sparse matrix that assigns each literal pair $(l_i, m_j) \in L \times M$ a substitution $\sigma_{i,j}$ such that $\sigma_{i,j}(l_i) = m_j$ or $\sigma_{i,j}(l_i) = \neg m_j$. In addition, a polarity $P_{i,j}$ is also assigned to (l_i, m_j), as follows: we set polarity $P_{i,j} = +$ if $\sigma_{i,j}(l_i) = m_j$ and $P_{i,j} = -$ if $\sigma_{i,j}(l_i) = \neg m_j$. This matrix is sparse because in general not all literal pairs $(l_i, m_j) \in L \times M$ can be matched. Additionally, it is again possible to prune (L, M) while filling the match set: if a row of the match set is empty, then there is some literal in L that cannot be matched to any literal in M. In this case, subsumption resolution cannot use L to simplify M, so the pair (L, M) is pruned.

SAT Solver. The *solver* of Algorithm 1 is the CDCL-based SAT solver introduced previously [10], which supports reasoning over matching substitutions in addition to standard propositional reasoning. This solver also features direct support for *AtMostOne* constraints. Solver performance was tuned for subsumption, which we retain for subsumption resolution. Each propositional variable v is associated with a substitution σ_v, and the solver ensures that all substitutions σ_v, for which v is assigned $\top$ in the current model, are compatible. Conceptually, a global substitution σ satisfying the invariant $\sigma = \bigcup\{\sigma_v \mid v = \top\}$ is kept in the SAT solver. In the following, we will write this binding as $v \Rightarrow \sigma_v \subseteq \sigma$.

Example 4.2. Suppose propositional variables v_1 and v_2 are associated with substitutions $\sigma_1 := \{x \mapsto y\}$ and $\sigma_2 := \{x \mapsto z\}$, respectively. As σ_1 and σ_2 are incompatible, the solver will block assigning $v_1 = \top$ and $v_2 = \top$ simultaneously since it would break the above invariant.

Encoding Constraints. Given the match set of (L, M), we formalise the subsumption resolution problem (2) as the conjunction of four constraints over matching substitutions. Our formalisation is given in Theorem 5.1 and is complete in the following sense: subsumption resolution can be applied over (L, M)

iff each constraint of Theorem 5.1 is satisfiable. Application of subsumption resolution is tested via satisfiability checking over our constraints from Theorem 5.1. Encodings of our subsumption resolution constraints are given in Sect. 5.

Building the Conclusion. If a model is found for the constraints encoding subsumption resolution, the conclusion $M \setminus \{m'\}$ of SR is built using the model.

5 Subsumption Resolution and SAT Encodings

As mentioned in Sect. 4, we turn the application of subsumption resolution SR over (L, M) into the satisfiability checking problem of Algorithm 1. We give our formalisation of SR in Theorem 5.1, followed by two encodings to SAT (Sect. 5.1–5.2) and adjustments to subsumption (Sect. 5.3).

Theorem 5.1 (Subsumption Resolution Constraints). *Clauses M and L are the main and side premise, respectively, of an instance of the subsumption resolution rule* SR *iff there exists a substitution σ that satisfies the following four properties:*

$$\textbf{\textit{existence}} \qquad \exists i\, j.\, \sigma(l_i) = \neg m_j \tag{3}$$

$$\textbf{\textit{uniqueness}} \qquad \exists j'.\, \forall i\, j.\, \big(\sigma(l_i) = \neg m_j \Rightarrow j = j'\big) \tag{4}$$

$$\textbf{\textit{completeness}} \qquad \forall i.\, \exists j.\, \big(\sigma(l_i) = \neg m_j \vee \sigma(l_i) = m_j\big) \tag{5}$$

$$\textbf{\textit{coherence}} \qquad \forall j.\, \big(\exists i.\, \sigma(l_i) = m_j \Rightarrow \forall i.\, \sigma(l_i) \neq \neg m_j\big) \tag{6}$$

We relate these constraints to the definition of subsumption resolution (2). The **existence** property (3) requires a literal m_j in M such that a literal l_i of L can be matched to $\neg m_j$, ensuring the existence of the resolution literal in SR. **Uniqueness** (4) asserts that the resolution literal m_j of SR is unique, required because SR performs only a single resolution step. **Completeness** (5) requires each literal in L be matched either to the complement of a resolution literal, or to a literal in M. Since each (complementary) literal in L is matched to one (resolution) literal of M, the completeness property ensures that the conclusion of SR subsumes M. Finally, **coherence** (6) states that all literals in M must be matched by literals in L with uniform polarity. This implies that all literals of L other than the resolution literal are present in the conclusion of SR. We note that these constraints can be used to recreate Example 3.1.

Example 5.1. The clause pair (L_2, M_2) of Fig. 1 does not satisfy the uniqueness property: both the match between $p(x_1)$ and $\neg p(y)$ and the match between $q(x_2)$ and $\neg q(c)$ are negative and so no substitution can satisfy all constraints simultaneously. Therefore, subsumption resolution cannot be applied over (L_2, M_2).

Example 5.2. The clause pair (L_3, M_3) violates the coherence property for all possible σ, since a negative map from $p(x_1)$ to $\neg p(y)$ cannot coexist with a positive map from $\neg p(x_2)$ to $\neg p(y)$. Subsumption resolution cannot be performed over (L_3, M_3).

5.1 Direct SAT Encoding of Subsumption Resolution

We present our encoding of subsumption resolution constraints as a SAT problem, allowing us to use Algorithm 1 for deciding the application of SR. In the sequel we consider the clauses L, M as in Theorem 5.1.

Compatibility. We introduce indexed propositional variables $b^+_{i,j}$ and $b^-_{i,j}$ to represent $\sigma(l_i) = m_j$ and $\sigma(l_i) = \neg m_j$ respectively, which we use to track compatible matching substitutions between literals of L and M. More precisely, a propositional variable is created if and only if the corresponding match is possible (i.e., in the formulas below, if no match exist, replace the corresponding propositional variable by $\bot$). As it is not possible to have simultaneously a substitution $\sigma_{i,j}(l_i) = m_j$ and $\sigma_{i,j}(l_i) = \neg m_j$, we also write $b_{i,j}$ to mean either $b^+_{i,j}$ or $b^-_{i,j}$ when the polarity of the match is irrelevant. Following Sect. 4, the variables are bound to their substitutions:

$$\textbf{SAT-based compatibility} \qquad \bigwedge_i \bigwedge_j [b_{i,j} \Rightarrow \sigma_{i,j} \subseteq \sigma] \tag{7}$$

SR **Constraints.** Constraints (3)–(6) of Theorem 5.1 employ *bounded* quantification over the finite number of literals in L, M. Expanding these quantifiers over their respective domains, we translate them into the following SAT formulas:

$$\textbf{SAT-based existence} \qquad \bigvee_i \bigvee_j b^-_{i,j} \tag{8}$$

$$\textbf{SAT-based uniqueness} \qquad \bigwedge_j \bigwedge_i \bigwedge_{i' \geq i} \bigwedge_{j' > j} \neg b^-_{i,j} \vee \neg b^-_{i',j'} \tag{9}$$

$$\textbf{SAT-based completeness} \qquad \bigwedge_i \bigvee_j b_{i,j} \tag{10}$$

$$\textbf{SAT-based coherence} \qquad \bigwedge_j \bigwedge_i \bigwedge_{i'} \neg b^+_{i,j} \vee \neg b^-_{i',j} \tag{11}$$

SR **as SAT Problem.** Based on the above, application of subsumption resolution is decided by the satisfiability of (7)∧(8)∧(9)∧(10)∧(11). This SAT formula extended with substitutions represents the result of *encodeConstraint*() in Algorithm 1 and is used further in Algorithm 3. When this formula is satisfiable, we construct the substitution σ required for SR by

$$\sigma = \bigcup \{\sigma_{i,j} \mid b_{i,j} = \top\}.$$

From the model of the SAT solver, we extract the first literal $b^-_{i,j}$ assigned $\top$, from which we conclude that the j^{th} literal in M is the resolution literal of SR. As such, application of SR over L and M results in replacing M by $M \setminus \{m_j\}$.

Remark 5.1. Implicitly, all l_i literals are mapped to at most one literal m_j. Indeed, if there were several literals m_j such that $\sigma(l_i) = m_j$ or $\sigma(l_i) = \neg m_j$, then either the respective matches are not compatible (guarded by the compatibility property (7)), there are identical literals in M, or M is a tautology (which is not allowed).

Remark 5.2. While we defined $b_{i,j}$ to be true if, and *only* if, $\sigma_{i,j} \subseteq \sigma$, we only encode the sufficient condition $b_{i,j} \Rightarrow \sigma_{i,j} \subseteq \sigma$. The completeness property (10) together with Remark 5.1 state that each l_i must have exactly one match to some m_j or $\neg m_j$. Therefore, if $\sigma_{i,j} \subseteq \sigma$ then the respective $b_{i,j}$ must be true and the condition also becomes necessary: $b_{i,j} \Leftarrow \sigma_{i,j} \subseteq \sigma$.

Example 5.3. Consider the pair (L_1, M_1) of Fig. 1. The match set ms of Algorithm 1 is:

$$\sigma_{i,j} = \begin{bmatrix} \{x_1 \mapsto g(y_1), x_2 \mapsto c\} & \{x_1 \mapsto f(c), x_2 \mapsto e\} \\ \bot & \{x_1 \mapsto c, x_2 \mapsto e\} \end{bmatrix} \quad P_{i,j} = \begin{bmatrix} + & - \\ & - \end{bmatrix}$$

Since $\sigma_{2,1}$ is incompatible with any substitution, $b_{2,1} = \bot$ need not be defined. This also allows to disregard SAT clauses that are trivially satisfied. The existence (8) and completeness (10) properties cannot have empty clauses: this is easily detected while filling the match set, and the instance of SR is pruned. Adding falsified literals in these constraints is unnecessary. The uniqueness (9) and coherence (11) properties have only negative polarity literals and therefore there is no need to add clauses containing $b_{2,1}$. In light of the previous comment, we use variables $b^+_{1,1}$, $b^-_{1,2}$ and $b^-_{2,2}$ and encode SR using the following constraints:

$$\begin{array}{ll}
b^+_{1,1} \Rightarrow \{x_1 \mapsto g(y_1), x_2 \mapsto c\} \subseteq \sigma & \textbf{SAT-based compatibility } \text{of } b^+_{1,1} \\
b^-_{1,2} \Rightarrow \{x_1 \mapsto f(c), x_2 \mapsto e\} \subseteq \sigma & \textbf{SAT-based compatibility } \text{of } b^-_{1,2} \\
b^-_{2,2} \Rightarrow \{x_2 \mapsto c, x_3 \mapsto e\} \subseteq \sigma & \textbf{SAT-based compatibility } \text{of } b^-_{2,2} \\
b^-_{1,2} \vee b^-_{2,2} & \textbf{SAT-based existence} \\
b^+_{1,1} \vee b^-_{1,2} & \textbf{SAT-based completeness}, i = 1 \\
b^-_{2,2} & \textbf{SAT-based completeness}, i = 2
\end{array}$$

The uniqueness (9) and coherence (11) properties are trivial here because the problem is simple: all $b^-_{i,j}$ have the same j, and no literal m_j can be mapped with different polarities. By using SAT solving from Algorithm 1 over the above SAT constraints, we obtain the SAT model $b^+_{1,1} \wedge \neg b^-_{1,2} \wedge b^-_{2,2}$, with $b^-_{2,2}$ the first literal assigned $\top$ with negative polarity. The application of SR over (L_1, M_1) yields the conclusion $M \setminus \{m_2\} = p(g(y_1), c)$, replacing M.

5.2 Indirect SAT Encoding of Subsumption Resolution

SAT-based formulas (9) and (11) may yield many constraints, with worst-case complexity $O(|L|^2|M|^2)$. In practice such situations rarely occur, since the match set ms is sparsely populated. Nevertheless, to alleviate this worst-case complexity, we further constrain the approach of Sect. 5.1. We introduce structuring propositional variables c_j such that c_j is $\top$ iff there exists a literal l_i with $\sigma(l_i) = \neg m_j$, which we encode as:

SAT-based structurality
$$\bigwedge_j \left[\neg c_j \vee \bigvee_i b^-_{i,j}\right] \wedge \bigwedge_j \bigwedge_i \left(c_j \vee \neg b^-_{i,j}\right) \tag{12}$$

SR as revised SAT problem. While the compatibility property (7) remains unchanged, the SR constrains of Theorem 5.1 are revised as given below.

SAT-based revised existence
$$\bigvee_j c_j \tag{13}$$

SAT-based revised uniqueness
$$\mathit{AtMostOne}(\{c_j, j = 1, ..., |M|\}) \tag{14}$$

SAT-based revised completeness
$$\bigwedge_i \bigvee_j b_{i,j} \tag{15}$$

SAT-based revised coherence
$$\bigwedge_j \bigwedge_i \left(\neg c_j \vee \neg b^+_{i,j}\right) \tag{16}$$

Similarly to Sect. 5.1, application of subsumption resolution is decided via Algorithm 1 by checking satisfiability of (7)$\wedge$ (12) $\wedge$ (13) $\wedge$ (14) $\wedge$ (15) $\wedge$ (16) . Using the above SAT formula as the result of *encodeConstraint*() in Algorithm 1, the worst-case behaviour is eliminated in exchange for $O(|M|)$ propositional variables, c_j. While the direct encoding of Sect. 5.1 is more efficient on small problems as it requires fewer variables and constraints, the indirect encoding of this section is expected to behave better on larger problems (see Sect. 7).

Remark 5.3. Note that the uniqueness property (14) is handled via *AtMostOne* constraints, based on the approach of [10]. If a variable c_j is set to $\top$, then our SAT *solver* in Algorithm 1 infers that all other variables $c_{j'}$ are set to $\bot$.

Example 5.4. Consider again the clause pair (L_1, M_1) of Fig. 1. Compared to Example 5.3, our revised encoding of SR requires one additional variable c_2, as m_2 in Example 5.3 is used with negative polarity. The revised constraints are:

$b^+_{1,1} \Rightarrow \{x_1 \mapsto g(y_1), x_2 \mapsto c\} \subseteq \sigma$	**SAT-based compatibility** of $b^+_{1,1}$
$b^-_{1,2} \Rightarrow \{x_1 \mapsto f(c), x_2 \mapsto e\} \subseteq \sigma$	**SAT-based compatibility** of $b^-_{1,2}$
$b^-_{2,2} \Rightarrow \{x_2 \mapsto c, x_3 \mapsto e\} \subseteq \sigma$	**SAT-based compatibility** of $b^-_{2,2}$
$\neg c_2 \vee b^-_{1,2} \vee b^-_{2,2}$	**SAT-based structurality** of c_2
$c_2 \vee \neg b^-_{1,2}$	**SAT-based structurality** of c_2
$c_2 \vee \neg b^-_{2,2}$	**SAT-based structurality** of c_2
c_2	**SAT-based revised existence**
$\mathit{AtMostOne}(\{c_2\})$	**SAT-based revised uniqueness**
$b^+_{1,1} \vee b^-_{1,2}$	**SAT-based revised completeness**, $i = 1$
$b^-_{2,2}$	**SAT-based revised completeness**, $i = 2$

The SAT solver returns $b^+_{1,1} \wedge \neg b^-_{1,2} \wedge b^-_{2,2} \wedge c_2$ as a solution to the above SAT problem, from which the application of SR yields a similar result to that of Example 5.3.

Remark 5.4. We note that our method naturally supports commutative predicates, such as equality. Let $\simeq$ denote object-level equality. Suppose we have literals $l_i := a \simeq b$ and $m_j := c \simeq d$. Two propositional variables with associated matching substitutions $\sigma_{i,j}$ and $\sigma'_{i,j}$ are introduced, where $\sigma_{i,j}$ matches $a \simeq b$ against $c \simeq d$ and $\sigma'_{i,j}$ matches $a \simeq b$ against $d \simeq c$. If zero or one matches exist, then the problem behaves exactly like the non-symmetric case. If both matches exist, then $\sigma_{i,j}$ and $\sigma'_{i,j}$ must be incompatible: otherwise, c and d would be identical terms and the trivial literal m_j would have been eliminated. Therefore, our SAT-based encodings for subsumption resolution do not need to be adapted and behave as expected.

5.3 SAT Constraints for Subsumption

In the new framework of Algorithm 1, the formulation suggested by [10] was adjusted to work with subsumption resolution. Algorithm 1 needs very little adaptation for subsumption: the *encodeConstraint*() method uses the encoding below, and the conclusion needs not be built as only the satisfiability of the formulas is relevant. The re-written SAT encoding becomes:

subsumption compatibility $$\bigwedge_i \bigwedge_j \left(b^+_{i,j} \Rightarrow \sigma_{i,j} \subseteq \sigma\right) \tag{17}$$

subsumption completeness $$\bigwedge_i \bigvee_j b^+_{i,j} \tag{18}$$

multiplicity conservation $$\bigwedge_j AtMostOne(\{b^+_{i,j}, i = 1, ..., |L|\}) \tag{19}$$

Note that the set of propositional variables used in our SAT-based formulas (17)–(19) encoding subsumption is a subset of the variables used by our SAT-based subsumption resolution constraints.

Pruning for Subsumption. The pruning technique described in Sect. 4 can be adapted into a stronger form for subsumption. In this case, we will check for multi-set inclusion between multi-sets of (predicates, polarity) pairs.

6 SAT-Based Subsumption Resolution in Saturation

In this section we discuss the integration of our SAT-based subsumption resolution approach within saturation-based proof search.

Forward/Backward Simplifications. For the purpose of efficient reasoning, saturation algorithms use two main variants of simplification inferences implementing redundancy. *Forward* simplifications are applied on a newly generated

Algorithm 2. SAT-based subsumption in saturation

```
$ms \leftarrow$ createMatchSet()
$solver \leftarrow$ createSatSolver($ms$)
procedure SUBSUMPTION($L, M$)
    $F_{S}$, $F_{SR} \leftarrow$ pruned($L, M$)
    ▷ $F_{S}$ (resp. $F_{SR}$) gets true if subsumption (resp. subsumption resolution) cannot succeed
    fillMatchSet($ms, L, M$)       ▷ Build the whole match set, and update $F_{S}$ and $F_{SR}$
    if $F_{S}$ then                 ▷ subsumption cannot be applied
        return NoSubsumption
    encodeConstraints($solver$, $ms$)      ▷ SAT-constraints of Section 5.3
    if $solver$.solve() is SAT then
        return Subsumed
    else
        return NoSubsumption
```

Algorithm 3. SAT-based subsumption resolution in saturation – with subsumption already set up via Algorithm 2

```
procedure SUBSUMPTIONRESOLUTION($L, M$)
                                   ▷ upon Algorithm 2 failing to subsume
                                   ▷ the match set is already set up
    if $F_{SR}$ then
        return NoSubsumptionResolution
    encodeConstraints($solver$, $ms$)     ▷ SAT constraints of Sect. 5.1 or Sect. 5.2
    if $solver$.solve() is SAT then
        return buildConclusion($solver$.getSolution(), $M$)    ▷ conclusion of subsumption resolution
    return NoSubsumptionResolution
```

clause M to check whether M can be simplified by an existing clause L. *Backward* simplifications use a newly generated clause L to check whether L can simplify existing clauses M. Backward simplification tends to be more expensive.

SAT-Based Subsumption Resolution in Saturation. Since subsumption is a stronger form of simplification, subsumption is checked before subsumption resolution. This means that subsumption resolution is applied only if subsumption fails for all candidate premises. We integrate Algorithm 1 within saturation so that it is used both for subsumption and subsumption resolution.

Algorithms 2–3 display a variation of the integration of our SAT-based approach for checking subsumption resolution during saturation. Since most of the setup of subsumption is also required for subsumption resolution, both simplification rules are set up at the same time. As such, whenever turning to subsumption resolution, the same match set ms from Algorithm 2 can be reused, while also taking advantage of pruning steps performed during subsumption.

We modified the forward simplification algorithm as described in Algorithm 4. In this new setting, checking the same pair (L, M) for subsumption

Algorithm 4. Forward simplification with SAT-based subsumption resolution

```
procedure ForwardSimplify($M, F$)
    $M^* \leftarrow$ NoSubsumptionResolution
    for $L \in F \setminus \{M\}$ do
        if subsumption($L, M$) is Subsumed then                ▷ using Algorithm 2
            $F \leftarrow F \setminus \{M\}$
            return $\top$                                  ▷ $M$ is subsumed and removed
        if $M^*$ = NoSubsumptionResolution then
            $M^* \leftarrow$ subsumptionResolution($L, M$)       ▷ using Algorithm 3
    if $M^* \neq$ NoSubsumptionResolution then
        $F \leftarrow F \setminus \{M\} \cup \{M^*\}$    ▷ $M^*$ is the conclusion of subsumption resolution
between $L$ and $M$
        return $\top$
    return $\bot$
```

Algorithm 5. Evaluation of SAT-based subsumption resolution

```
procedure ForwardSimplifyWrapper($M, F$)
    $s \leftarrow$ startTimer()
    $r \leftarrow$ ForwardSimplify($M, F$)                    ▷ Benchmarked method
                                                    ▷ Prevent modification of $F$
    $e \leftarrow$ endTimer()
    writeInFile($e - s$)
    $r' \leftarrow$ Oracle($M, F$)
    checkCoherence($r, r'$)                               ▷ Empiric check
    return $r'$
```

directly followed by subsumption resolution enables us to use Algorithms 2 and 3 efficiently. Algorithm 4 pays the price of checking subsumption resolution even if subsumption may succeed, but in practice inefficiencies in this respect are seen rarely.

Role of Indices. When applying inferences that require terms or literals to unify or match, modern automated first-order theorem provers typically use *term indices* [9] to consider only viable candidates within the set of clauses. Subsumption and subsumption resolution is no exception. Our testbed system Vampire currently uses a substitution tree to index clauses for matching by their literals (Sect. 7).

7 Implementation and Experiments

We implemented and integrated our SAT-based subsumption resolution approach in the saturation-based first-order theorem prover Vampire [6][1].

[1] https://github.com/vprover/vampire/tree/robin_c-subsumption_resolution.

Versions compared. We use following versions of Vampire in our evaluation:

- Vampire$_M$ is the *master* branch without SAT-based subsumption resolution;
- Vampire$_I$ is the SAT-based subsumption resolution with the *indirect* encoding of Sect. 5.2 and a standard forward simplification algorithm with Algorithm 1 — that is, Algorithm 4 is not used here;
- Vampire$_I^*$ uses the *indirect* encoding with Algorithms 2–4;
- Vampire$_D^*$ uses the *direct* encoding of Sect. 5.1 and Algorithms 2–4.

Experimental Setting. To evaluate our work, we used the examples of the TPTP library (version 8.1.2) [15]. In our evaluation, 24 926 problems were used out of the 25 257 TPTP problems; the remaining problems are not supported by Vampire (e.g., problems with both higher-order operators and polymorphism).

Our experimental evaluation was done on a machine with two 32-core AMD Epyc 7502 CPUs clocked at 2.5 GHz and 1006 GiB of RAM (split into 8 memory nodes of 126 GiB shared by 8 cores). Each benchmark problem was run with the options `-sa otter -t 60`, meaning that we used the Otter saturation algorithm [7] with a 60-second time-out. We use the Otter strategy because it is the most aggressive in terms of simplification and therefore runs the most subsumption resolutions. We turned off the AVATAR framework (`-av off`) in order to have full control over SAT-based reasoning in Vampire.

Evaluation Setup. Our evaluation process is summarised in Algorithm 5, incorporating the following notes.

- The conclusion clause of the subsumption resolution rule `SR` is not necessarily unique. Therefore, different versions of subsumption resolution, including our work based on direct and indirect SAT encodings, may not return the same conclusion clause of `SR`. Hence, applying different versions of subsumption resolution over the same clauses may change the saturation process.
- Saturation with our SAT-based subsumption resolution takes advantage of subsumption checking (see Algorithms 3 and 4). Therefore, only checking subsumption resolution on pairs of clauses is not a fair nor viable comparison, as isolating subsumption checks from subsumption resolution is not what we aimed for (due to efficiency).
- CPU cache influences results. For example, two consecutive runs of Algorithm 4 may be up to 25% faster on second execution, due to cache effects.

For the reasons above, we decided to measure the run time of a complete execution of Algorithm 4. To prevent the branches to change, an `Oracle` is used to choose the path to follow. The `Oracle` is based on our indirect SAT encoding (Vampire$_I^*$). This way, the same computation graph is used for all evaluated methods. To prevent cache preheating, we run the `Oracle` after the respective evaluated method. This way the cache is in a normal state for the evaluated method. To measure the run time of Algorithm 4, a `Wrapper` method was built on top of the `Forward Simplify` procedure of Algorithm 4. This `Wrapper` replaces the `Forward Simplify` loop in Vampire with minimal changes to the code. To empirically verify the correctness of our results, we used the `Wrapper` to compare the result of the evaluated method with the result of the `Oracle`.

Experimental Details and Analysis. Fig. 2 lists the cumulative instances solved by the respective VAMPIRE versions, highlighting the strength of forward simplifications for effective saturation.

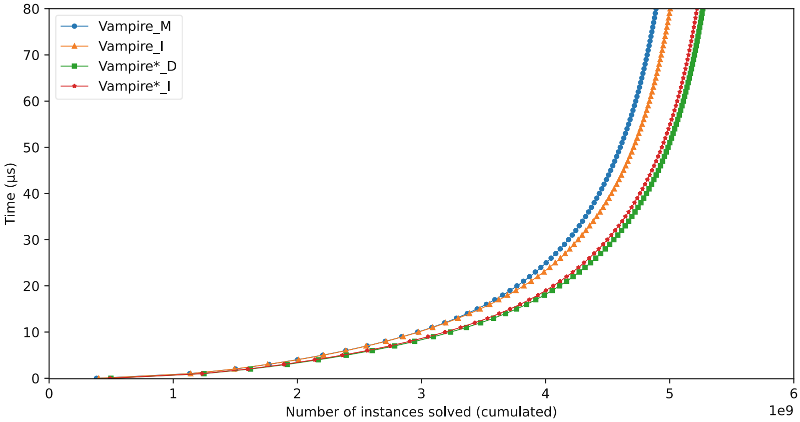


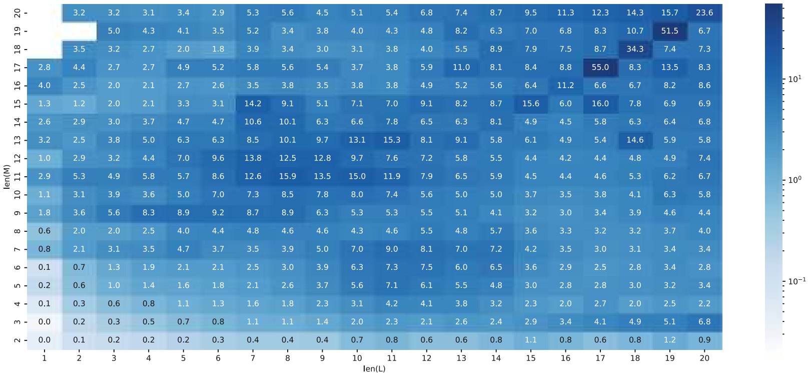
Fig. 2. Cumulative instances of applying subsumption resolution, using the TPTP examples. A point (n, t) on the graph means that n forward simplify loops were executed in less than t μs. The flatter the curve, the faster the VAMPIRE version is.

Table 1. Average time spent in the `Forward Simplify` loop. VAMPIRE_D^* is the fastest method, closely followed by the VAMPIRE_I^*. However, the indirect encoding is much more stable and has a lower variance.

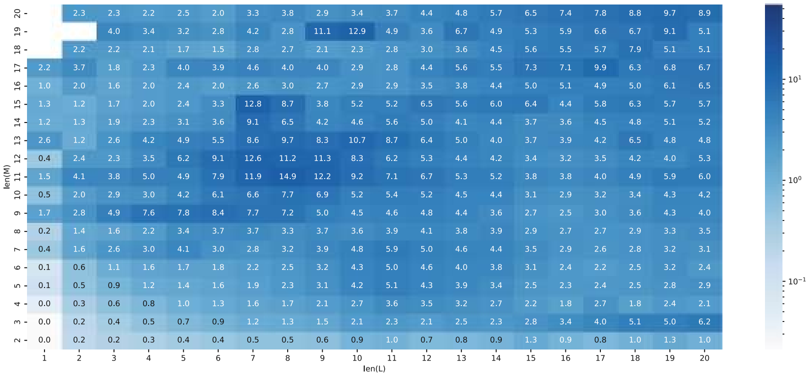
Prover	Average	Std. Dev.	Speedup
VAMPIRE_M	42.63 μs	1609.06 μs	0%
VAMPIRE_I	40.13 μs	1554.52 μs	6.2%
VAMPIRE_D^*	34.39 μs	1047.85 μs	23.9%
VAMPIRE_I^*	34.55 μs	250.25 μs	23.4%

Remark 7.1. Our experimental summary in Fig. 2 shows that the total number of `Forward Simplify` loops ran in 60 s. However, the average and standard deviation were computed only on the intersection of the problems solved. That is, only the `Forward Simplify` loops finished by all the methods are taken into account. Otherwise, if a hard problem is solved in, for instance, 1 000 000 μs by one method, and times out for another, the average for the better would increase a lot, but the weaker method would not be penalised. Table 1 summarises the average solving time of our evaluation.

Comparison of Encodings. We correlated the constraint building and SAT solving time with the length of clauses, using the different encodings of Sects. 5.1–5.2. Figure 3 shows that on larger clauses, the average computation time increases faster for the direct encoding than for the indirect encoding.



(a) Average time (μs) for creating/solving direct encoding constraints (Section 5.1).



(b) Average time (μs) for creating/solving indirect encoding constraints (Section 5.2).

Fig. 3. Average time (μs) spent on the creating and solving SAT-based subsumption resolution constraints.

Table 2. Number of TPTP problems solved by the considered versions of Vampire. The run was made using the options `-sa otter -av off` with a timeout of 60 s. The **Gain/Loss** column reports the difference of solved instances compared to Vampire$_M$.

Prover	Total Solved	Gain/Loss
Vampire$_M$	10 555	baseline
Vampire$^*_D$	10 667	(+141, −29)
Vampire$^*_I$	10 658	(+133, −30)

Experimental Summary. Our experiments show that Vampire$^*_I$ yields the most stable approach for SAT-based subsumption resolution (Table 1), especially when it comes on solving large instances (Fig. 3). Our results demonstrate the superiority of SAT-based subsumption resolution used with forward simplifications in saturation (e.g., VAMPIRE$^*_D$ and VAMPIRE$^*_I$), as concluded by Table 2.

8 Conclusion

We advocate SAT solving for improving saturation-based first-order theorem proving. We encode powerful simplification rules, in particular subsumption resolution, as SAT problems, triggering eager and efficient reasoning steps for the purpose of keeping proof search small. Our experiments with VAMPIRE showcase the benefit of SAT-based subsumption. In the future, we aim to further extend simplification rules with SAT solving, in particular focusing on subsumption demodulation for equality reasoning [3].

Acknowledgements. This work is the result of a research internship hosted at TU Wien and defended at the University of Liège. The authors would like to thank Pascal Fontaine for valuable discussions and comments. We acknowledge funding from the ERC Consolidator Grant ARTIST 101002685 and the FWF SFB project SpyCoDe F8504.

References

1. Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. J. Log. Comput. **4**(3), 217–247 (1994)
2. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Handbook of Automated Reasoning, vol. 2, pp. 19–99. Elsevier and MIT Press (2001)
3. Gleiss, B., Kovács, L., Rath, J.: Subsumption demodulation in First-order theorem proving. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12166, pp. 297–315. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_17
4. Kapur, D., Narendran, P.: NP-Completeness of the set unification and matching problems. In: IJCAR, pp. 489–495 (1986)
5. Korovin, K.: Inst-Gen – a modular approach to instantiation-based automated reasoning. In: Programming Logics: Essays in Memory of Harald Ganzinger, pp. 239–270 (2013)
6. Kovács, L., Voronkov, A.: First-Order theorem proving and VAMPIRE. In: CAV, pp. 1–35 (2013)
7. McCune, W., Wos, L.: Otter– the CADE-13 competition incarnations. J. Autom. Reason. **18**, 211–220 (1997)
8. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Handbook of Automated Reasoning, pp. 371–443. Elsevier and MIT Press (2001)
9. Ramakrishnan, I.V., Sekar, R., Voronkov, a.: Term indexing. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 2, pp. 1853–1964. Elsevier and MIT Press (2001)
10. Jakob Rath, Armin Biere, and Laura Kovács. First-Order Subsumption via SAT Solving. In FMCAD, page 160, 2022
11. Reger, G., Suda, M.: The uses of SAT solvers in vampire. In: Vampire Workshop, pp. 63–69 (2015)

12. Reger, G., Suda, M.: Global subsumption revisited (Briefly). In: Vampire @ IJCAR, pp. 61–73 (2016)
13. Schulz, S.: Simple and efficient clause subsumption with feature vector indexing. In: Automated Reasoning and Mathematics - Essays in Memory of William W. McCune, pp. 45–67 (2013)
14. Schulz, S., Cruanes, S., Vukmirovic, P.: Faster, higher, stronger: E 2.3. In: CADE, pp. 495–507 (2019)
15. Sutcliffe, G.: The TPTP problem library and associated infrastructure. From CNF to TH0, TPTP v6.4.0. J. Autom. Reason. **59**(4), 483–502 (2017)
16. Tammet, T.: Towards efficient subsumption. In: CADE, pp. 427–441 (1998)
17. Voronkov, A.: AVATAR: the architecture for First-Order theorem provers. In: CAV, pp. 696–710 (2014)
18. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: CADE, pp. 140–145 (2009)

A More Pragmatic CDCL for IsaSAT and Targetting LLVM (Short Paper)

Mathias Fleury[1,2,3](✉) and Peter Lammich[4,5]

[1] University Freiburg, Freiburg, Germany
fleury@cs.uni-freiburg.de
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[3] Johannes Kepler University Linz, Linz, Austria
[4] University of Manchester, Manchester, UK
[5] University of Twente, Twente, Netherlands
p.lammich@utwente.nl

Abstract. IsaSAT is the most advanced verified SAT solver, but it did not yet feature inprocessing (to simplify and strengthen clauses). In order to improve performance, we enriched the base calculus to not only do CDCL but also inprocess clauses. We also replaced the target of our code synthesis by Isabelle/LLVM. With these improvements, we can solve 4 times more SAT Competition 2022 problems than the original IsaSAT version, and 4.5 times more problems than any other verified SAT solver we are aware of. Additionally, our changes significantly reduce the trusted code base of our verification.

1 Introduction

SAT solving is a very important tool that has been extensively used in various applications like mathematics or cryptography. To ensure the correctness of the answer provided by a SAT solver, there are two approaches: either producing a certificate that can be checked independently or verifying a SAT solver. The first approach has been extensively studied and works very well in practice [19,26,28] – only checked proofs are counted in the SAT Competition [2].

The second approach, i.e., verifying a whole SAT solver is orders of magnitudes more complex than checking a certificate. To this end, the goal of the IsaFoL (Isabelle Formalization of Logic) [3] effort is to develop methodology and libraries for formalizing modern research in automated reasoning. In this context, we have verified a CDCL calculus (conflict-driven clause learning) and a two-watched literals data structure (Sect. 2). To show that they are useful, we have developed the verified SAT solver IsaSAT [8], which we later optimized [12]. To our surprise, it won the EDA Challenge 2021 defeating all the non-verified solvers, but, as expected, it finished last at the SAT Competition 2022 [2]. However, the former used a much shorter timeout (200 s, not announced before the competition) whereas the latter uses 5000 s.

In this paper, we present our new developments in IsaSAT, which make our solver arguably the most advanced formally verified SAT solver to date: inprocessing and verifying fast LLVM code [20] rather than slow functional code.

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 207–219, 2023.
https://doi.org/10.1007/978-3-031-38499-8_12

Inprocessing is a critical feature of modern SAT solvers (e.g., every winner of the SAT Competition since 2013 includes it). In order to use it in our formally verified solver, we had to extend our verified CDCL calculus: Our new PCDCL calculus includes features to encompass various inprocessing techniques, even if we have not yet implemented every possible technique (Sect. 3).

We generate IsaSAT by exporting a model in the interactive theorem prover Isabelle [22] to executable code. Earlier we used Isabelle's default code generator to export to Standard ML (SML). However, the performance was not sufficient – especially memory consumption was very high. Thus, we switched to Isabelle/LLVM [18], which generates LLVM intermediate representation (LLVM IR). Apart from allowing faster imperative code, it also reduced the trusted code base (Sect. 4), replacing the rather niche MLton [27] compiler by only the backend of the widely used LLVM.

Porting our entire development to Isabelle/LLVM required some changes and some cleanup. Moreover, when we implemented and verified inprocessing, we realized that some design decisions need to be improved. In Sect. 5, we report on our experiences and lessons learned while porting and extending IsaSAT.

Finally, we have benchmarked IsaSAT on the problems from the SAT Competition 2022. We show that just porting IsaSAT from SML to Isabelle/LLVM significantly improved the performance, and the new inprocessing techniques combined with heuristic improvements give us another significant increase, demonstrating the usefulness of our PCDCL calculus (Sect. 6).

This presentation is an extended version of our (non-peer-reviewed) system description from the EDA Challenge 2021 [13] and the SAT Competition 2022 [6]. Compared to that version, we have provided much more details on PCDCL, our experience porting the development to LLVM, and performance tests.

2 Preliminaries

CDCL. CDCL is a procedure that builds a partial assignment called *a trail* either by guessing (called *deciding*) or propagating information. If the partial assignment is a model, the algorithm stops. If there is a conflict between the partial assignment and a clause, the partial assignment is repaired and a new clause is learned. For more details (beyond the scope of this paper), we refer the reader to the *Handbook of Satisfiability* [7].

We use a transition system for our formalization of CDCL [8]. Its state consists of the trail M, the (multi)sets of initial and learned clauses (N and U), and the conflict clause to analyze (or None if there is none). We show one rule, decide, that adds L to the current assignment M:

```
inductive decide :: $'st \Rightarrow 'st \Rightarrow bool$ where
  undefined_lit $M$ $L \Longrightarrow |L| \in |N| \Longrightarrow$
  decide $(M, N, U, \mathsf{None})$ $(L \cdot M, N, U, \mathsf{None})$
```

If no conflict has been found so far (None), we add the new literal L at the beginning of the trail M. We prove that our set of rules is terminating and correct [8].

Code Synthesis. To generate the IsaSAT code, we start from the abstract rules like `decide` and gradually refine it to some deterministic functions using the Refinement Framework [16]. Then, we rely on Sepref [17] to synthesize code: It takes an (Isabelle) function and synthesizes a new version, replacing functional data structures (like lists) by imperative data structures (like arrays). There are two versions of the tool. The older version, which we used before [8,12], uses Imperative HOL [9] and Isabelle's standard (trusted) code generator [14] to export code into various functional languages. We used Standard ML (SML) with the compiler MLton [27], because it offers (by far) the best performance for our use case. The new Sepref is part of the Isabelle/LLVM library (developed by the second author) and generates LLVM IR from a model of LLVM IR inside the theorem prover. The code generator interprets a shallow embedding of Isabelle/LLVM as equivalent to similar looking LLVM code. This reduces the trusted code base in two ways: first, the trusted pretty printer is simpler, and, second, instead of the rather niche full compiler MLton, we use only the backend of the widely used LLVM [20].

The biggest difference is that Imperative HOL allows arbitrary large arrays and integers, whereas Isabelle/LLVM is more realistic, requiring integers (in particular array offsets, see Sect. 5.1) to have a fixed bit-width.

Related Work. Our goal is to produce a fully verified SAT solver, without any runtime checks, that both *terminates* and returns a *correct model* while using efficient data structures. No other solver achieves all three goals. The SAT solver TrueSAT from Andrici and Ciobaca [1] relies on the original DPLL and uses less efficient data structures (including counters instead of watch lists), but it terminates. Historically, this would roughly correspond to SAT solver from the early 90s. However, it only uses stateless heuristics, and it is not clear if the approach can be extended to CDCL (where the solver learns and keeps new clauses) or to stateful heuristics (like VSIDS [21]). The solvers versat [23] and Creusat [25] go into a similar direction with CDCL instead of DPLL, but prove a weaker correctness property: they only show that an UNSAT result is correct, while a SAT result requires an additional check. Also, termination is not proved. Only proving this partial property makes many proofs considerably easier, in particular adding restarts. Oe et al's solver versat [23] was the first partially verified solver that could run benchmarks from the SAT Competition. More recently, Skotåm [25] has verified in his Master's thesis the SAT solver CreuSAT using the Creusot framework (relying on Why3 internally). While CreuSAT is much faster than versat in our tests, its correctness relies on (trusted) SMT solvers, and the proofs are not checked by a small kernel like our Isabelle code. However, the verification also takes much less time (a few minutes compared to several hours).

Modern SAT solvers use inprocessing to make the subsequent CDCL run heuristically faster [15]. In particular, clauses are strengthened and global transformation (e.g., to remove variables) are applied. Two techniques (that we do not support), variable elimination and addition, slowly change the models of the formula by changing the set of variable. The SAT solver then reconstructs a model of the original formula at the end. Fazekas et al. [11] made it compatible with incremental SAT solving. All others inprocessing technique fit into our extended CDCL described in the next section.

3 Pragmatic CDCL for Inprocessing

SAT solvers nowadays apply a combination of CDCL (most of the time) and inprocessing (sometimes). Therefore, we extended our calculus similarly. At the core, we have our terminating CDCL. We also allow for formula transformation and restarts. We call the combination *pragmatic CDCL* or PCDCL.

Splitting the Clause Set. Inprocessing makes it possible to strengthen and simplify clauses. However, we want models from the final set of clauses to remain models from the initial set of clauses. Deleting clauses is not possible: if we start with the clauses $A \vee C$ and $B \vee \neg B$, removing the tautology means that the model A of $A \vee C$ is not a model of the initial clause set anymore. Hence we want to keep the literal B without considering the tautology for propagation/conflict.

To solve the issue we split our set of clauses into two parts: clauses that are useful for propagation and clauses that can be ignored but are kept for their literals. Thus we keep the set of all literals $\mathcal{A}$ constant. For our proof of refinement to the original CDCL, we have to make sure that the new behavior is also possible in the original calculus – in particular we do not miss propagations or conflicts. In the case of tautologies, this is simple (they are never used). If we consider subsumption, like $A \vee B$ subsumes $A \vee B \vee C$, whenever the latter propagates, then the former is a conflict. Therefore, the behavior is compatible.

While the idea of splitting our clauses seems surprising, the additional clause sets are only required for the connection to our CDCL transition system, and we entirely remove them when generating the code. Moreover, the refinement is easier as we do not have to update our heuristics to remove literals (and potentially shorten arrays). Finally, this is similar to the behavior of SAT solvers like Kissat [4]: while the clauses are removed, all literals of the problem are set.

In our original refinement, we have split the clauses to distinguish between clauses of length 1 (where we cannot distinguish two distinct literals and thus they cannot fit into our two-watched literals data structures) and longer clauses, but the aim was only distinguishing on the length.

One important point to notice is that the role of our clause sets changes. In our original CDCL, N was the (immutable) set of initial clauses and U contains the redundant clauses that can be removed at any point: N ensures that we do gain new models during our transformations. Now, the set changes: strengthening an irredundant clause from N also shortens the clause that is in there. Therefore, a naive version could remove literals.

Overall we have 4 sets of clauses: the irredundant clauses N and the redundant U clauses, and each one is divided into the active clauses (N_a and U_a) and the inactive (discarded) clauses (N_d and U_d). For example, tautologies or subsumed clauses are discarded, but remain in N, so literals are never removed. In our development there are actually three sets (containing a literal set at level 0 or tautologies, subsumed clauses, and false clauses) to reduce the number of case distinction in some proofs. We never demote irredundant clauses to redundant ones, but we can promote them.

Inprocessing Rules. Our aim when picking the rule is to be general (like we can learn any useful clause) and then we specialize rules to specific techniques. We will show this with the example of subsumption-resolution [7]. When doing subsumption-resolution, we resolve two clauses together if the conclusion is shorter. Then we can remove either one or both of the antecedents. For example, resolving $A \vee B \vee C$ with $A \vee \neg C$ produces the clause $A \vee B$ with subsumes the former clause. If the latter clause was $A \vee B \vee \neg C$, the resolved clauses would actually subsume both clauses.

One of the most important inprocessing rule learns any possible clause. To simplify the presentation, we will only give the rules operating on the learned clauses, but similar rules exists for the initial set of clauses.

$$
\begin{array}{l}
\texttt{inductive}\ \mathsf{cdcl\_learn\_clause} :: {}'\mathit{prag\_st} \Rightarrow {}'\mathit{prag\_st} \Rightarrow \mathit{bool}\ \texttt{where} \\
\quad |C| \subseteq |N + N_d| \Longrightarrow \mathsf{count\_decided}\, M = 0 \Longrightarrow \\
\quad N \wedge N_d \vDash C \Longrightarrow \neg\mathsf{tautology}\, C \Longrightarrow \mathsf{distinct}\, C \Longrightarrow \\
\quad \mathsf{cdcl\_learn\_clause}\ (M, N, U, \mathsf{None}, N_d, U_d) \\
\qquad\qquad\qquad\qquad (M, N, U \wedge C, \mathsf{None}, N_d, U_d)
\end{array}
$$

The side conditions not only include that the clause is entailed and duplicate-free, but also the clause is not a tautology and we do not break CDCL invariants ($\mathsf{count\_decided}\, M = 0$). Then we can deactivate subsumed clauses:

$$
\begin{array}{l}
\texttt{inductive}\ \mathsf{cdcl\_subsumed} :: {}'\mathit{prag\_st} \Rightarrow {}'\mathit{prag\_st} \Rightarrow \mathit{bool}\ \texttt{where} \\
\quad C \subseteq D \Longrightarrow \mathsf{count\_decided}\, M = 0 \Longrightarrow \\
\quad \mathsf{cdcl\_subsumed}\ (M, N, U \wedge C \wedge D, \mathsf{None}, N_d, U_d) \\
\qquad\qquad\qquad\qquad (M, N, U \wedge C, \mathsf{None}, N_d, D \wedge U_d)
\end{array}
$$

We combine these rules to express subsumption-resolution: We first learn the clause obtained by resolution. Then we can remove the antecedents. If either antecedent is in N, we also have promoted the conclusion from N to U. The advantage of our approach is that we can express other inprocessing techniques without adding new rules, only by specializing them.

Overall we have 9 rules with some overlap with CDCL (propagation and conflict), but mostly simplification of clauses (removing true clauses and false literals from clauses) and pure literal deletion: When a literal always appears positively (or always negatively), we can set this literal to be true unconditionally (later removing all clauses containing it): every model after adding the clause is also a model of the original set of clauses but not the opposite. This is the first transformation that does not preserve models in IsaSAT or any other verified SAT solvers.

Refinement of Subsumption-Resolution. While the definition of subsumption resolution is very simple, the refinement to code was challenging.

We verified forward subsumption [7] following CaDiCaL [5] (unbounded however, so all clauses selected heuristically are checked). We sort clauses by size and check if the current candidate is subsumed by one of the smaller clauses. Because we use two-watched literals, we need to distinguish between the binary clauses (than can produce new units) and the other clauses. At the end, we implemented two forward subsumption passes: one for binary clauses only and the other for larger clauses.

To subsume the candidates, we build occurrence lists and populate them with binary clauses, whereas Kissat [5] reuses watch lists. Moreover, for efficiency, we need a new marking data structure for efficient detection of subsuming-resolution.

4 Correctness of the Code and Completeness

Our specification $\mathsf{model\_if\_satisfiable}$ takes the multiset of clauses and returns a model (if there is one) or None if the clauses are unsatisfiable. Our implementation $\mathsf{IsaSAT_{SML}\ opts}$ takes an array containing the clauses and returns an optional array containing the assignment, assuming that the clauses do not contain duplicated literals or the empty clause (precondition $\mathsf{proper\_lits\_no\_dups\_\bot}$). The additional argument opts activates and deactivates certain techniques for solving. The following theorem states that our implementation refines the specification:

Theorem 1 (SML End-to-End Correctness). *The following refinement relation holds:*

$$\begin{aligned}&(\mathsf{IsaSAT_{SML}\ opts},\ \mathsf{model\_if\_satisfiable})\\ \in\ &[\mathsf{proper\_lits\_no\_dups\_\bot}]\,\mathsf{clauses\_assn} \rightarrow \mathsf{option\_model\_assn}\end{aligned}$$

The LLVM version is nearly the same. It can handle duplicated literals and the empty clause. Moreover, the new specification $\mathsf{model\_if\_satisfiable\_bounded}$ allows for an *unknown* result if arrays would grow larger than the size permitted by the fixed bit-width. While this limit does not exist in Imperative_HOL, it exists in practice as no machine supports arrays that large. Therefore, we technically weakened our theorem, but did not change practical guarantees on the generated code. For $\mathsf{IsaSAT_{SML}}$ we start [12] with 64-bit unsigned integers and only switch to GMP integers if the arrays grow too large.

Theorem 2 (LLVM End-to-End Correctness). *The following refinement relation holds:*

$$\begin{aligned}&(\mathsf{IsaSAT_{LLVM}\ opts},\ \mathsf{RETURN} \circ \mathsf{model\_if\_satisfiable\_bounded})\\ \in\ &[\mathsf{proper\_lits}]\,\mathsf{clauses\_assn} \rightarrow \mathsf{option\_model\_assn}\end{aligned}$$

Moreover, the change from SML to LLVM reduces the trusted code base: The Isabelle/LLVM model is closer to the actual LLVM, such that the trusted pretty-printer is simpler. LLVM is also more low-level, such that fewer parts of the compiler have to be trusted. Finally, the LLVM compiler is more widely used and tested than the rather niche MLton compiler we used before.

5 Experience Porting the Development to LLVM

We report on the challenges we faced when updating the huge IsaSAT formalization (Sect. 5.1). Moreover, we report on the unverified parts of IsaSAT (Sect. 5.2), and finally compile some lessons learned (Sect. 5.3).

5.1 Required Changes

Before porting the development to LLVM, we removed our only remaining source of unbounded integers: the clause indices during the garbage collection. As garbage collection does not happen very often, we did not expect this to make a difference. Surprisingly, it turns out to have a performance impact.

Isabelle/LLVM is an entire tool set, including a fork of the original Sepref tool. While related to the original Sepref tool, there are different libraries, and the development of the two versions has diverged.

Initially, we tried to support both versions of Sepref. We ended up with two sets of files for code synthesis, and duplication of some libraries (to provide constants defined in Isabelle/LLVM but not in Sepref$_{\mathrm{SML}}$). This significantly complicated our refinement approach, although we made it conceptually cleaner during the porting. Then, we realized that IsaSAT$_{\mathrm{LLVM}}$ was much faster than IsaSAT$_{\mathrm{SML}}$ (we observed a factor 2 on our test files), and decided to discontinue the SML backend.

With this, also some workarounds for SML specific performance issues (like the tuple `uint32 * bool * uint64` being much less efficient than combining the uint32 and the Boolean into a single 64-bit number) became obsolete.

Compilation. We have experimented with compilation flags before to improve performance. We know from the SML code that we need to increase the level of inlining, because many small functions make the verification easier. The same applies for LLVM and the easiest solution is to use link-time optimization that increases the inlining level as a side effect. However, this makes profiling impossible – exactly like the SML code. So there is no regression here.

Tuples. In 2021, we observed a major performance regression of the synthesis, caused by a new feature in Sepref$_{\mathrm{LLVM}}$: pointer-equality tracking caused quadratic behaviour for case-splits of tuples. As our solver state is a large tuple, synthesis became impossible (several dozen minutes for simple functions).

To avoid the issue, we decided to work around on the abstract level, using getter and setter functions for the state's components, rather than case splitting. Now, every function on the state would first get the required components, update them, and then put them back. For example:

```
definition rescore_conflict :: clause_index ⇒ isasat ⇒ isasat where
  rescore_conflict C S = do{
    let (M, S) = extract_trail S;
    ... (*reads the trail M and can change it*) ...
    let S = update_trail M S;
    RETURN S
  }
```

This makes synthesis much faster. However, the ownership model of Sepref does not allow aliasing, nor do our refinement relations allow leaving a 'gap' in the state where we moved out an element. As an easy work-around, we resorted to placing dummy-values, like empty lists, in the state, hoping that LLVM would optimize away the allocations and deallocations for these values. However, this did not happen: In the hot-spot of the SAT solver, the propagation loop, the dummy value for the trail was recreated and freed each time. Thus, we locally resorted to unfolding our code to make sure that we need only one free in the inner propagation loop. We leave a more principled solution of this problem (possibly changing Sepref) to future work.

We even attempted to go one step further (as the state-of-the-art SAT solver Kissat [4] does) and simply passing a pointer to the state structure as argument. Once we had already changed our refinement with accessors, we simply had to change them to work on a pointer. However, we never managed to make the synthesized code efficient. We observed a factor of 10 slower code. Hand-optimizing the accessors (basically making sure that LLVM understands that we care only about one component) reduced this to factor 2 slower. Once we realized that the LLVM optimizer was replacing the pointer by the structure passed directly as argument, we gave up on that approach.

5.2 Unverified Parts

In the generated SAT solver, there are some parts that we cannot verify. First, the parser is not verified, because the file system has no model in Isabelle (unlike CakeML, where conditions apply however). To this end, we link the verified code with an unverified C program, which provides the parser and command line interface.

Second, Isabelle/LLVM does not support any output (like statistics, or the DRAT proofs [28] required for the SAT Competition). For the SML version, we could use a feature of Isabelle's code generator to (axiomatically) implement a function by some external function (e.g. a function that does nothing in the model, by a printing function). As Isabelle/LLVM does not yet have such a feature, we resorted to post-processing the generated code (i.e., a function that does nothing in the model, is replaced by a printing function or even a function storing some literals for DRAT proofs). Note that this post-processing is not required for IsaSAT to work (but it won't print DRAT proofs).

5.3 Lessons Learned

Lesson 1: Embrace Duplication. We have already highlighted the importance of the set of all possible literals $\mathcal{A}$, in particular to establish a bound on the size of various arrays. At first, we tried to avoid duplicating this set across the different components on the specification side. This, however, resulted in a closer coupling of the various refinement proofs, impeding modularity: data structures that, conceptually, are just a small part of the whole state, have to be formalized on the whole state, just to have the set $\mathcal{A}$ available. We solved this problem by duplicating the set $\mathcal{A}$ on the abstract level for all new data structures. Note that this duplication is removed in a later refinement stage.

Lesson 2: The Limits are Isabelle Files. Checking our Isabelle files takes nearly two hours. This can be explained by three factors: 1. the heuristic and code synthesis amounts to 91 000 loc, making it a very large formalization; 2. the synthesis is single-threaded (for technical reasons); 3. Sepref encourages a style that is not very parallel: every refinement starts with a call to a tactic `refine_vcg` that generates the goals (meaning that all successive tactics have to wait). To improve performance we have attempted [12] to generate more standard proofs in Isar (by generating the text corresponding to the theorems to prove), but it is not clear that this style is faster as huge number of variables are generated (this style is required for more complicated proofs, however).

In order to improve Isabelle's performance and speed-up the testing of new heuristics in $\mathsf{IsaSAT}_{\mathsf{LLVM}}$, we have split the files into three parts: the shared definitions of the functions to refine, the (single-threaded) synthesis, and the correctness proof of the refinement. Even with these optimizations, proof checking still takes 2 h. There is also no clear improvement path. The old SML version has a similar problem, but it is overall faster because it has fewer features, making it less critical.

Lesson 3: Performance Bugs exist. In order to improve performance, we need to measure and observe performance. To solve that problem, IsaSAT prints statistics and produces some timing information. The statistics during the run made identifying scheduling bugs for the different techniques possible – we accidentally ran some techniques way too often or barely ever. Especially because we increase the interval between two inprocessing rounds geometrically, a simple statistics at the end of the run is not sufficient. One interesting performance bug we found was that we accidentally inverted reducing clauses (marking them as removed) and garbage collection (physically removing them). Therefore, we would nearly always physically delete clauses. We never saw this issue, because we also printed the statistics inverted. To help debugging performance, we produce some timing information by measuring time in the C program:

```
c propagate          : 83.48% (581.66 s)
c reduce             : 0.12% (0.82 s)
c subsumption        : 0.06% (0.39 s)
c pure_lits          : 0.05% (0.33 s)
```

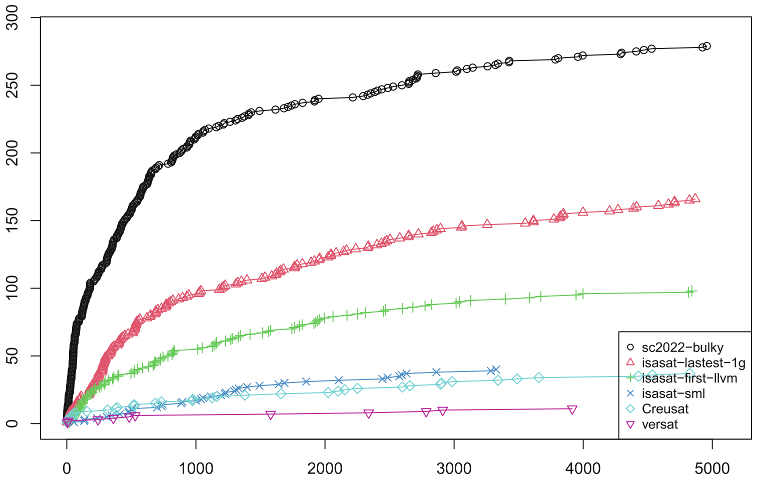


Fig. 1. CDF of the performance of SAT solvers

```
c binary_simp          : 0.02% (0.15 s)
c GC                   : 0.16% (1.10 s)
```

This helps to identify bottlenecks but also outliers where one technique is particularly slow and requires some limits or a change in the scheduling to avoid slowing down the solver too much. This makes it possible to identify errors like allocations in loops. The overall timing matches what we expect from other SAT solvers (although usually they spend more time on inprocessing and less on propagation).

6 Performance

In order to study the performance we have run 3 different IsaSAT versions: the original SML solver (using MLton with the LLVM backend), the first port of the IsaSAT solver, and the current version with inprocessing and various other improvements on heuristics that do not require any change on our PCDCL calculus, notably rephasing and target phases [10] (but no local search) and the alternation between aggressive restarts (heuristically seems better for UNSAT) and few restarts (seems better for SAT) following the ideas of Chanseok Oh [24].

We run all the benchmarks from the SAT Competition 2022 on an Intel Xeon E5-2620 v4 CPU at 2.10 GHz (with turbo-mode disabled) with a memory limit of 7 GB and a timeout of 5000 s. For comparison, we have included VERSAT [23] and CreuSAT [25]. For completeness, we have included Kissat [6] (more precisely the bulky version submitted for the anniversary track).

The results are given in Fig. 1 as a CDF (the higher the curve, the more solved problems). The first surprise is that CreuSAT performs similarly to IsaSAT$_{\text{SML}}$ (37 vs 40 solved problems), worse than expected given the results reported in the

Master's thesis [25] that tested on the 2015 benchmarks. We suspect that is due to the garbage collection and the fact that problems from the SAT Competition have become harder.

There is a clear improvement when going from the SML version to the LLVM version (98 solved), while the latest version solves 166. The SML version produces 335 out-of-memory errors (OOMs), the base LLVM version is more memory efficient (23 OOMs) like the latest IsaSAT version (19 OOMs) or CaDiCaL that has the same memory layout (17 OOMs). However, there is still a large gap to reach the performance level of Kissat and its inprocessing techniques.

7 Conclusion

We have reported on updating our verified SAT solver IsaSAT to a more powerful base calculus (our pragmatic CDCL) which can express inprocessing, and to the more efficient Isabelle/LLVM backend. We have also compiled important lessons learned from proof-engineering and maintaining large formalizations like IsaSAT (∼200 kloc of proofs).

Our changes made IsaSAT solve 4 times more problems (166/40), making it the most efficient verified SAT solver. At the same time, our verification is more complete than the next fastest verified solvers.

Most techniques (including the two most important, vivification and probing) either fit into our new PCDCL base calculus or do not require any change (like random walk [10] that is conjectured to be the reason for the major performance improvement in 2020). One major technique that we cannot currently express is variable elimination, because models are changed and need to be fixed. We leave the required extensions to our PCDCL for future work.

Acknowledgments. The work presented here was done over several years and several work places. The first author was supported for some time by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), and the LIT AI Lab funded by the State of Upper Austria. We thank the anonymous reviewers for their detailed comments.

References

1. Andrici, C.C., Ciobâcă, S.: A verified implementation of the DPLL algorithm in Dafny. Mathematics **10**(13), 1–26 (2022). https://ideas.repec.org/a/gam/jmathe/v10y2022i13p2264-d850381.html
2. Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, Department of Computer Science, University of Helsinki, Finland (2022)
3. Becker, H., et al.: IsaFoL: Isabelle formalization of logic. https://bitbucket.org/isafol/isafol/
4. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proceedings of SAT

Competition 2020 - Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)

5. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2021. In: Proceedings of the SAT Competition 2021 - Solver and Benchmark Descriptions (2021). submitted
6. Biere, A., Fleury, M.: Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In: Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proceedings of the SAT Competition 2022 - Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2022-1, pp. 10–11. University of Helsinki (2022)
7. Biere, A., Järvisalo, M., Kiesl, B.: Preprocessing in SAT solving. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications. 2nd edn, vol. 336, pp. 391–435. IOS Press (2021). https://doi.org/10.3233/FAIA200992
8. Blanchette, J.C., Fleury, M., Lammich, P., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. J. Autom. Reason. **61**(1–4), 333–365 (2018). https://doi.org/10.1007/s10817-018-9455-7
9. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_14
10. Cai, S., Zhang, X., Fleury, M., Biere, A.: Better decision heuristics in CDCL through local search and target phases. J. Artif. Intell. Res. **74**, 1515–1563 (2022). https://doi.org/10.1613/jair.1.13666
11. Fazekas, K., Biere, A., Scholl, C.: Incremental inprocessing in SAT solving. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 136–154. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_9
12. Fleury, M.: Optimizing a Verified SAT Solver. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2019. LNCS, vol. 11460, pp. 148–165. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20652-9_10
13. Fleury, M.: IsaSAT and Kissat entering the EDA Challenge 2021 (2021). https://www.eda-ai.org/results/, system description accepted at the EDA Challenge 2021. https://m-fleury.github.io/ox-hugo/Fleury-EDA-Challenge-2021.pdf
14. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12251-4_9
15. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 355–370. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_28
16. Lammich, P.: Automatic data refinement. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 84–99. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_9
17. Lammich, P.: Refinement to imperative HOL. J. Autom. Reason. **62**(4), 481–503 (2017). https://doi.org/10.1007/s10817-017-9437-1
18. Lammich, P.: Generating verified LLVM from Isabelle/HOL. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving, ITP 2019, 9–12, September 2019, Portland, OR, USA. LIPIcs, vol. 141, pp. 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.22

19. Lammich, P.: Efficient verified (UN)SAT certificate checking. J. Autom. Reason. **64**(3), 513–532 (2019). https://doi.org/10.1007/s10817-019-09525-z
20. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004, pp. 75–88. IEEE (2004). https://doi.org/10.1109/cgo.2004.1281665
21. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, 18–22 June 2001, pp. 530–535. ACM (2001). https://doi.org/10.1145/378239.379017
22. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
23. Oe, D., Stump, A., Oliver, C., Clancy, K.: versat: a verified modern SAT solver. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 363–378. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27940-9_24
24. Oh, C.: Between SAT and UNSAT: the fundamental difference in CDCL SAT. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 307–323. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_23
25. Skotåm, S.H.: CreuSAT - using rust and Creusot to create the world's fastest deductively verified SAT solver. Master's thesis, University of Oslo (2022). https://www.duo.uio.no/handle/10852/96757
26. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: `cake_lpr`: verified propagation redundancy checking in CakeML. In: TACAS 2021. LNCS, vol. 12652, pp. 223–241. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_12
27. Weeks, S.: Whole-program compilation in MLton. In: Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, 16 September 2006, p. 1. ACM Press (2006). https://doi.org/10.1145/1159876.1159877
28. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31

Proving Non-Termination by Acceleration Driven Clause Learning (Short Paper)

Florian Frohn(✉) and Jürgen Giesl(✉)

LuFG Informatik 2, RWTH Aachen University, Aachen, Germany
`florian.frohn@cs.rwth-aachen.de`, `giesl@informatik.rwth-aachen.de`

Abstract. We recently proposed *Acceleration Driven Clause Learning* (ADCL), a novel calculus to analyze satisfiability of *Constrained Horn Clauses* (CHCs). Here, we adapt ADCL to transition systems and introduce ADCL-NT, a variant for disproving termination. We implemented ADCL-NT in our tool LoAT and evaluate it against the state of the art.

1 Introduction

Termination is one of the most important properties of programs, and thus termination analysis is a very active field of research. Here, we are concerned with *dis*proving termination of *transition systems* (TSs), a popular intermediate representation for verification of programs written in more expressive languages.

Example 1. Consider the following TS $\mathcal{T}$ with entry-point init and two further *locations* ℓ_1, ℓ_2 over the variables x, y, z, where x', y', z' represent the values of x, y, z *after* applying a transition, and $\overset{=}{x}$, $x{+}{+}$, and $x{-}{-}$ abbreviate $x' = x$, $x' = x + 1$, and $x' = x - 1$. The first two transitions are a variant[1] of `chc-LIA-Lin_052` from the *CHC Competition '22* [7] and the last two are a variant[2] of `flip2_rec.jar-obl-8` from the *Termination and Complexity Competition (TermComp)* [21].

$$\mathsf{init} \to \ell_1 \, [\![x' \leq 0 \land z' \geq 5000 \land y' \leq z']\!] \tag{τ_{i}}$$

$$\ell_1 \to \ell_1 \, [\![y \leq 2 \cdot z \land x{+}{+} \land ((x < z \land \overset{=}{y}) \lor (x \geq z \land y{+}{+})) \land \overset{=}{z}]\!] \tag{τ_{ℓ_1}}$$

$$\ell_1 \to \ell_2 \, [\![x = y \land x > 2 \cdot z \land \overset{=}{x} \land \overset{=}{y}]\!] \tag{$\tau_{\ell_1 \to \ell_2}$}$$

$$\ell_2 \to \ell_2 \, [\![x = y \land x > 0 \land \overset{=}{x} \land y{-}{-}]\!] \tag{$\tau_{\ell_2}^{=}$}$$

$$\ell_2 \to \ell_2 \, [\![x > 0 \land y > 0 \land x' = y \land ((x > y \land y' = x) \lor (x < y \land \overset{=}{y}))]\!] \tag{$\tau_{\ell_2}^{\neq}$}$$

[1] We generalized the example to make it more interesting, and we added the condition $y \leq 2 \cdot z$ to enforce termination of τ_{ℓ_1}.

[2] We combined the transitions for the cases $x > y$ and $x < y$ into the equivalent transition $\tau_{\ell_2}^{\neq}$ to demonstrate how our approach can deal with disjunctions in conditions.

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2).

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 220–233, 2023.
https://doi.org/10.1007/978-3-031-38499-8_13

At ℓ_1, $\mathcal{T}$ operates in two "phases": First, just x is incremented until x reaches z (1^{st} disjunct of τ_{ℓ_1}). Then, x and y are incremented until y reaches $2 \cdot z + 1$ (2^{nd} disjunct of τ_{ℓ_1}). If $x = y = c$ holds for some $c > 1$ at that point (which is the case if $x \leq y = z$ holds initially), then the execution can continue at ℓ_2 as follows:

$$\ell_2(c, c, c_z) \longrightarrow_{\tau_{\ell_2}^{=}} \ell_2(c, c-1, c_z) \longrightarrow_{\tau_{\ell_2}^{\neq}} \ell_2(c-1, c, c_z) \longrightarrow_{\tau_{\ell_2}^{\neq}} \ell_2(c, c, c_z) \longrightarrow_{\tau_{\ell_2}^{=}} \ldots$$

Here, $\ell_2(c, c, c_z)$ means that the current location is ℓ_2 and the values of x, y, z are c, c, c_z. The 1^{st} and 2^{nd} step with $\tau_{\ell_2}^{\neq}$ satisfy the 1^{st} ($x > y \land \ldots$) and 2^{nd} ($x < y \land \ldots$) disjunct of $\tau_{\ell_2}^{\neq}$'s condition, respectively. Thus, $\mathcal{T}$ does not terminate.

Example 1 is challenging for state-of-the-art tools for several reasons. First, more than 5000 steps are required to reach ℓ_2, so reachability of ℓ_2 is difficult to prove for approaches that unroll the transition relation or use other variants of iterative deepening. Thus, `chc-LIA-Lin_052` is beyond the capabilities of most other state-of-the-art tools for proving reachability.

Second, the pattern "$\tau_{\ell_2}^{=}$, 1^{st} disjunct of $\tau_{\ell_2}^{\neq}$, 2^{nd} disjunct of $\tau_{\ell_2}^{\neq}$" must be found to prove non-termination. Therefore, `flip2_rec.jar-obl-8` (which does not use disjunctions) cannot be solved by other state-of-the-art termination tools.

Third, Example 1 contains disjunctions, which are not supported by many termination tools. Presumably, the reason is that most techniques for (dis)proving termination of loops are restricted to conjunctions (e.g., due to the use of templates and Farkas' Lemma). While disjunctions can be avoided by splitting disjunctive transitions according to the DNF of their conditions, this leads to an exponential blow-up in the number of transitions.

We present an approach that can prove non-termination of systems like Example 1 automatically. To this end, we tightly integrate non-termination techniques into our recent *Acceleration Driven Clause Learning (ADCL)* calculus [16], which has originally been designed for CHCs, but it can also be used to analyze TSs.

Due to the use of acceleration techniques that compute the transitive closure of recursive transitions, ADCL finds long witnesses of reachability automatically. If acceleration techniques cannot be applied, it unrolls the transition relation, so it can easily detect complex patterns of transitions that admit non-terminating runs. Finally, ADCL reduces reasoning about disjunctions to reasoning about conjunctions by considering conjunctive variants of disjunctive transitions. Thus, combining ADCL with non-termination techniques for conjunctive transitions allows for disproving termination of TSs with complex Boolean structure.

After introducing preliminaries in Sect. 2, Sect. 3 presents a straightforward adaption of ADCL to TSs. Section 4 introduces our main contribution: ADCL-NT, a variant of ADCL for proving non-termination. Finally, in Sect. 5, we discuss related work and demonstrate the power of our approach by comparing it with other state-of-the-art tools. All proofs can be found in [19].

2 Preliminaries

We assume familiarity with basics from many-sorted first-order logic. $\mathcal{V}$ is a countably infinite set of variables and $\mathcal{A}$ is a first-order theory over a k-sorted signature $\Sigma_{\mathcal{A}}$ with carrier $\mathcal{C}_{\mathcal{A}} = (\mathcal{C}_{\mathcal{A},1}, \ldots, \mathcal{C}_{\mathcal{A},k})$. $\mathsf{QF}(\Sigma_{\mathcal{A}})$ is the set of all quantifier-free first-order formulas over $\Sigma_{\mathcal{A}}$, which are w.l.o.g. assumed to be in negation normal form, and $\mathsf{QF}_{\wedge}(\Sigma_{\mathcal{A}})$ only contains conjunctions of $\Sigma_{\mathcal{A}}$-literals. Given a first-order formula η over $\Sigma_{\mathcal{A}}$, σ is a *model* of η (written $\sigma \models_{\mathcal{A}} \eta$) if it is a model of $\mathcal{A}$ with carrier $\mathcal{C}_{\mathcal{A}}$, extended with interpretations for $\mathcal{V}$ such that η is satisfied. As usual, $\models_{\mathcal{A}} \eta$ means that η is valid, and $\eta \equiv_{\mathcal{A}} \eta'$ means $\models_{\mathcal{A}} \eta \iff \eta'$.

We write $\vec{x}$ for sequences and x_i is the i^{th} element of $\vec{x}$. We use "::" for concatenation of sequences, where we identify sequences of length 1 with their elements, so we may write, e.g., $x :: xs$ instead of $[x] :: xs$.

Transition Systems. Let $d \in \mathbb{N}$ be fixed, and let $\vec{x}, \vec{x}' \in \mathcal{V}^d$ be disjoint vectors of pairwise different variables. Each $\psi \in \mathsf{QF}(\Sigma_{\mathcal{A}})$ induces a relation $\longrightarrow_{\psi}$ on $\mathcal{C}_{\mathcal{A}}^d$ where $\vec{s} \longrightarrow_{\psi} \vec{t}$ iff $\psi[\vec{x}/\vec{s}, \vec{x}'/\vec{t}]$ is satisfiable. So for the condition $\psi := (x = y \wedge x > 0 \wedge \overset{=}{x} \wedge y{-}{-})$ of $\tau_{\ell_2}^{=}$, we have $(4,4,4) \longrightarrow_{\psi} (4,3,7)$. $\mathcal{L} \supseteq \{\mathsf{init}, \mathsf{err}\}$ is a finite set of *locations*. A *configuration* is a pair $(\ell, \vec{s}) \in \mathcal{L} \times \mathcal{C}_{\mathcal{A}}^d$, written $\ell(\vec{s})$. A *transition* is a triple $\tau = (\ell, \psi, \ell') \in \mathcal{L} \times \mathsf{QF}(\Sigma_{\mathcal{A}}) \times \mathcal{L}$, written $\ell \to \ell' \llbracket \psi \rrbracket$, and its *condition* is $\mathsf{cond}(\tau) := \psi$. W.l.o.g., we assume $\ell \neq \mathsf{err}$ and $\ell' \neq \mathsf{init}$. Then τ induces a relation $\longrightarrow_{\tau}$ on configurations where $\mathfrak{s} \longrightarrow_{\tau} \mathfrak{t}$ iff $\mathfrak{s} = \ell(\vec{s}), \mathfrak{t} = \ell'(\vec{t})$, and $\vec{s} \longrightarrow_{\psi} \vec{t}$. So, e.g., $\ell_2(4,4,4) \longrightarrow_{\tau_{\ell_2}^{=}} \ell_2(4,3,7)$. We call τ *recursive* if $\ell = \ell'$, *conjunctive* if $\psi \in \mathsf{QF}_{\wedge}(\Sigma_{\mathcal{A}})$, *initial* if $\ell = \mathsf{init}$, and *safe* if $\ell' \neq \mathsf{err}$. Moreover, we define $(\ell \to \ell' \llbracket \psi \rrbracket)|_{\psi'} := \ell \to \ell' \llbracket \psi' \rrbracket$. A *transition system* (TS) $\mathcal{T}$ is a finite set of transitions, and it induces the relation $\longrightarrow_{\mathcal{T}} := \bigcup_{\tau \in \mathcal{T}} \longrightarrow_{\tau}$.

Chaining $\tau = \ell_s \to \ell_t \llbracket \psi \rrbracket$ and $\tau' = \ell_s' \to \ell_t' \llbracket \psi' \rrbracket$ yields $\mathsf{chain}(\tau, \tau') := (\ell_s \to \ell_t' \llbracket \psi_c \rrbracket)$ where $\psi_c := \psi[\vec{x}'/\vec{x}''] \wedge \psi'[\vec{x}/\vec{x}'']$ for fresh $\vec{x}'' \in \mathcal{V}^d$ if $\ell_t = \ell_s'$, and $\psi_c := \bot$ (meaning *false*) if $\ell_t \neq \ell_s'$. So $\longrightarrow_{\mathsf{chain}(\tau,\tau')} = \longrightarrow_{\tau} \circ \longrightarrow_{\tau'}$, and $\mathsf{chain}(\tau_{\ell_1 \to \ell_2}, \tau_{\ell_2}^{=}) = \ell_1 \to \ell_2 \llbracket \psi \rrbracket$ where $\psi \equiv_{\mathcal{A}} (x = y \wedge x > 2 \cdot z \wedge x > 0 \wedge \overset{=}{x} \wedge y{-}{-})$. For non-empty, finite sequences of transitions we define $\mathsf{chain}([\tau]) := \tau$ and $\mathsf{chain}([\tau_1, \tau_2] :: \vec{\tau}) := \mathsf{chain}(\mathsf{chain}(\tau_1, \tau_2) :: \vec{\tau})$. We lift notations for transitions to finite sequences via chaining. So $\mathsf{cond}(\vec{\tau}) := \mathsf{cond}(\mathsf{chain}(\vec{\tau}))$, $\vec{\tau}$ is *recursive* if $\mathsf{chain}(\vec{\tau})$ is recursive, $\longrightarrow_{\vec{\tau}} = \longrightarrow_{\mathsf{chain}(\vec{\tau})}$, etc. If τ is initial and $\mathsf{cond}(\tau :: \vec{\tau}) \not\equiv_{\mathcal{A}} \bot$, then $(\tau :: \vec{\tau}) \in \mathcal{T}^+$ is a *finite run*. $\mathcal{T}$ is safe if every finite run is safe. If there is a σ such that $\sigma \models_{\mathcal{A}} \mathsf{cond}(\vec{\tau}')$ for every finite prefix $\vec{\tau}'$ of $\vec{\tau} \in \mathcal{T}^{\omega}$, then $\vec{\tau}$ is an *infinite run*. If no infinite run exists, then $\mathcal{T}$ is *terminating*.

Acceleration. *Acceleration techniques* compute the transitive closure of relations. In the following definition, we only consider relations defined by conjunctive formulas, since many existing acceleration techniques do not support disjunctions [4], or have to resort to approximations in the presence of disjunctions [13].

Definition 2 (Acceleration). *An* acceleration technique *is a function* $\mathsf{accel} : \mathsf{QF}_{\wedge}(\Sigma_{\mathcal{A}}) \mapsto \mathsf{QF}_{\wedge}(\Sigma_{\mathcal{A}'})$ *such that* $\longrightarrow_{\psi}^{+} = \longrightarrow_{\mathsf{accel}(\psi)}$, *where* $\mathcal{A}'$ *is a first-order theory. For recursive conjunctive transitions* τ, *we define* $\mathsf{accel}(\tau) := \tau|_{\mathsf{accel}(\mathsf{cond}(\tau))}$.

So we clearly have $\longrightarrow_{\tau}^{+} = \longrightarrow_{\mathsf{accel}(\tau)}$. Note that most theories are not "closed under acceleration". E.g., accelerating the Presburger formula $x_1' = x_1 + x_2 \land \overset{=}{x}_2$ yields the non-linear formula $n > 0 \land x_1' = x_1 + n \cdot x_2 \land \overset{=}{x}_2$. If neither $\mathbb{N}$ nor $\mathbb{Z}$ are contained in $\mathcal{C}_{\mathcal{A}}$, then an additional sort for the range of n is required in the formula that results from applying accel. Hence, Definition 2 allows $\mathcal{A}' \neq \mathcal{A}$.

3 ADCL for Transition Systems

We originally proposed the ADCL calculus to analyze satisfiability of linear *Constrained Horn Clauses* (CHCs) [16]. Here, we rephrase it for TSs, and in Sect. 4, we modify it for proving non-termination. The adaption to TSs is straightforward as TSs can be transformed into equivalent linear CHCs and vice versa (see, e.g., [10]).

To bridge the gap between transitions τ where $\mathsf{cond}(\tau) \in \mathsf{QF}(\Sigma_{\mathcal{A}})$ and acceleration techniques for formulas from $\mathsf{QF}_{\land}(\Sigma_{\mathcal{A}})$, ADCL uses *syntactic implicants.*

Definition 3 (Syntactic Implicants [16, Def. 6]). *If* $\psi \in \mathsf{QF}(\Sigma_{\mathcal{A}})$*, then:*

$$\begin{aligned}
\mathsf{sip}(\psi, \sigma) &:= \bigwedge \{\pi \textit{ is a literal of } \psi \mid \sigma \models_{\mathcal{A}} \pi\} && \textit{if } \sigma \models_{\mathcal{A}} \psi \\
\mathsf{sip}(\psi) &:= \{\mathsf{sip}(\psi, \sigma) \mid \sigma \models_{\mathcal{A}} \psi\} \\
\mathsf{sip}(\tau) &:= \{\tau|_{\psi} \mid \psi \in \mathsf{sip}(\mathsf{cond}(\tau))\} && \textit{for transitions } \tau \\
\mathsf{sip}(\mathcal{T}) &:= \bigcup_{\tau \in \mathcal{T}} \mathsf{sip}(\tau) && \textit{for TSs } \mathcal{T}
\end{aligned}$$

Here, sip *abbreviates* syntactic implicant projection.

As $\mathsf{sip}(\psi, \sigma)$ is restricted to literals from ψ, $\mathsf{sip}(\psi)$ is finite. Syntactic implicants ignore the semantics of literals. So we have, e.g., $(X > 1) \notin \mathsf{sip}(X > 0 \land X > 1) = \{X > 0 \land X > 1\}$. It is easy to show $\psi \equiv_{\mathcal{A}} \bigvee \mathsf{sip}(\psi)$, and thus $\longrightarrow_{\mathcal{T}} = \longrightarrow_{\mathsf{sip}(\mathcal{T})}$.

Since $\mathsf{sip}(\tau)$ is worst-case exponential in the size of $\mathsf{cond}(\tau)$, we do not compute it explicitly. Instead, ADCL constructs a run $\vec{\tau}$ step by step, and to perform a step with τ, it searches for a model σ of $\mathsf{cond}(\vec{\tau} :: \tau)$. If such a model exists, it appends $\tau|_{\mathsf{sip}(\mathsf{cond}(\tau), \sigma)}$ to $\vec{\tau}$. This corresponds to a step with a conjunctive variant of τ whose condition is satisfied by σ. In other words, our calculus constructs $\mathsf{sip}(\mathsf{cond}(\tau), \sigma)$ "on the fly" when performing a step with τ, where $\sigma \models_{\mathcal{A}} \mathsf{cond}(\vec{\tau} :: \tau)$

The core idea of ADCL is to learn new, *non-redundant* transitions via acceleration. Essentially, a transition is redundant if its transition relation is a subset of another transition's relation. Thus, redundant transitions are not useful for (dis-)proving safety.

Definition 4 (Redundancy, [16, Def. 8]). *A transition* τ *is* (strictly) redundant *w.r.t.* τ'*, denoted* $\tau \sqsubseteq \tau'$ *(*$\tau \sqsubset \tau'$*) if* $\longrightarrow_{\tau} \subseteq \longrightarrow_{\tau'}$ *(*$\longrightarrow_{\tau} \subset \longrightarrow_{\tau'}$*). For a TS* $\mathcal{T}$*, we have* $\tau \sqsubseteq \mathcal{T}$ *(*$\tau \sqsubset \mathcal{T}$*) if* $\tau \sqsubseteq \tau'$ *(*$\tau \sqsubset \tau'$*) for some* $\tau' \in \mathcal{T}$*.*

In the sequel, we assume oracles for redundancy, satisfiability of $\mathsf{QF}(\Sigma_{\mathcal{A}})$-formulas, and acceleration. In practice, we use incomplete techniques instead (see Sect. 5).

From now on, let $\mathcal{T}$ be the TS that is being analyzed with ADCL. A *state* of ADCL consists of a TS $\mathcal{S}$ that augments $\mathcal{T}$ with *learned transitions*, a run $\vec{\tau}$ of $\mathcal{S}$ called the *trace*, and a sequence of sets of *blocking transitions* $[B_i]_{i=0}^{k}$, where transitions that are redundant w.r.t. B_k must not be appended to the trace.

The following definition introduces the ADCL calculus. It extends the trace step by step (using the rule Step, which performs an evaluation step with a transition) and learns new transitions via acceleration (Accelerate) whenever a suffix of the trace is recursive. To avoid non-terminating ADCL-derivations, our notion of *redundancy* from Definition 4 is used to backtrack whenever a suffix of the trace corresponds to a special case of another (learned) transition (Covered). Moreover, Backtrack is used whenever a run cannot be continued. A more detailed explanation of ADCL is provided after Definition 5.

Definition 5 (ADCL [16, Def. 9, 10]). *A* state *is a triple* $(\mathcal{S}, [\tau_i]_{i=1}^{k}, [B_i]_{i=0}^{k})$ *where* $\mathcal{S} \supseteq \mathcal{T}$ *is a TS,* $\bigcup_{i=0}^{k} B_i \subseteq \mathsf{sip}(\mathcal{S})$*, and* $[\tau_i]_{i=1}^{k} \in \mathsf{sip}(\mathcal{S})^*$*. The transitions in* $\mathsf{sip}(\mathcal{T})$ *are called* original *and the transitions in* $\mathsf{sip}(\mathcal{S}) \setminus \mathsf{sip}(\mathcal{T})$ *are* learned*. A transition* $\tau_{k+1} \sqsubseteq B_k$ *is* blocked*, and* $\tau_{k+1} \not\sqsubseteq B_k$ *is* active *if* $\mathsf{chain}([\tau_i]_{i=1}^{k+1})$ *is an initial transition with satisfiable condition (i.e.,* $[\tau_i]_{i=1}^{k+1}$ *is a run). Let*

$$\mathsf{bt}(\mathcal{S}, [\tau_i]_{i=1}^{k}, [B_0, \dots, B_k]) := (\mathcal{S}, [\tau_i]_{i=1}^{k-1}, [B_0, \dots, B_{k-1} \cup \{\tau_k\}])$$

where bt *abbreviates "backtrack". Our calculus is defined by the following rules.*

$$\frac{}{\mathcal{T} \rightsquigarrow (\mathcal{T}, [], [\varnothing])} \quad (\textsc{Init}) \qquad \frac{\tau \in \mathsf{sip}(\mathcal{S}) \text{ is active}}{(\mathcal{S}, \vec{\tau}, \vec{B}) \rightsquigarrow (\mathcal{S}, \vec{\tau} :: \tau, \vec{B} :: \varnothing)} \quad (\textsc{Step})$$

$$\frac{\vec{\tau}^{\circlearrowleft} \text{ is recursive} \quad |\vec{\tau}^{\circlearrowleft}| = |\vec{B}^{\circlearrowleft}| \quad \mathsf{accel}(\vec{\tau}^{\circlearrowleft}) = \tau \not\sqsubseteq \mathsf{sip}(\mathcal{S})}{(\mathcal{S}, \vec{\tau} :: \vec{\tau}^{\circlearrowleft}, \vec{B} :: \vec{B}^{\circlearrowleft}) \rightsquigarrow (\mathcal{S} \cup \{\tau\}, \vec{\tau} :: \tau, \vec{B} :: \{\tau\})} \quad (\textsc{Accelerate})$$

$$\frac{\vec{\tau}' \sqsubset \mathsf{sip}(\mathcal{S}) \quad \text{or} \quad \vec{\tau}' \sqsubseteq \mathsf{sip}(\mathcal{S}) \wedge |\vec{\tau}'| > 1}{s = (\mathcal{S}, \vec{\tau} :: \vec{\tau}', \vec{B}) \rightsquigarrow \mathsf{bt}(s)} \quad (\textsc{Covered})$$

$$\frac{\text{all transitions from } \mathsf{sip}(\mathcal{S}) \text{ are inactive} \quad \tau \text{ is safe}}{s = (\mathcal{S}, \vec{\tau} :: \tau, \vec{B}) \rightsquigarrow \mathsf{bt}(s)} \quad (\textsc{Backtrack})$$

$$\frac{\vec{\tau} \text{ is unsafe}}{(\mathcal{S}, \vec{\tau}, \vec{B}) \rightsquigarrow \mathsf{unsafe}} \quad (\textsc{Refute}) \qquad \frac{\text{all transitions from } \mathsf{sip}(\mathcal{S}) \text{ are inactive}}{(\mathcal{S}, [], [B]) \rightsquigarrow \mathsf{safe}} \quad (\textsc{Prove})$$

We write $\overset{\mathrm{I}}{\rightsquigarrow}$, $\overset{\mathrm{S}}{\rightsquigarrow}$, ... to indicate that the rule Init, Step, ... was used. Step adds a transition to the trace. When the trace has a recursive suffix, Accelerate allows for learning a new transition which then replaces the recursive suffix on the trace, or we may backtrack via Covered if the recursive suffix is redundant. Note that Covered does not apply if $\vec{\tau}' \sqsubseteq \mathsf{sip}(\mathcal{S})$ and $|\vec{\tau}'| = 1$, as it could immediately undo every Step, otherwise. If no further Step is possible, Backtrack applies. Note that Backtrack and Covered block the last transition from the trace so that we do not perform the same Step again. If $\vec{\tau}$ is an unsafe run, Refute yields unsafe, and if the entire search space has been exhausted without finding an unsafe run (i.e., if all initial transitions are blocked), Prove yields safe.

The definition of ADCL in [16] is more liberal than ours: In our setting, Accelerate may only be applied if the learned transition is non-redundant, and our definition of "active transitions" enforces that the first transition on the trace is always an initial transition. In [16], these requirements are not enforced by the definition of ADCL, but by the definition of *reasonable strategies* [16, Def. 14]. For simplicity, we integrated these requirements into Definition 5. Additionally, Covered should be preferred over Accelerate, and Accelerate should be preferred over Step.

Example 6. We apply ADCL to a version of Example 1 with the additional transition

$$\ell_1 \to \mathsf{err}\, [\![x = y \wedge x > 2 \cdot z \wedge \overline{\overline{x}} \wedge \overline{\overline{y}} \wedge \overline{\overline{z}}]\!]. \qquad (\tau_{\mathsf{err}})$$

$$\begin{aligned}
\mathcal{T} \overset{\mathrm{I}}{\rightsquigarrow}\ & (\mathcal{T}, [], [\varnothing]) \overset{\mathrm{S}}{\rightsquigarrow}{}^{2} (\mathcal{T}, [\tau_{\mathsf{i}}, \tau_{\ell_1}|_{\psi_{x<z}}], [\varnothing, \varnothing, \varnothing]) && (x \leq 1 \wedge z \geq 5k \wedge y \leq z) \\
\overset{\mathrm{A}}{\rightsquigarrow}\ & (\mathcal{S}_1, [\tau_{\mathsf{i}}, \tau^{+}_{x<z}], [\varnothing, \varnothing, \{\tau^{+}_{x<z}\}]) && (x \leq z \wedge z \geq 5k \wedge y \leq z) \\
\overset{\mathrm{S}}{\rightsquigarrow}\ & (\mathcal{S}_1, [\tau_{\mathsf{i}}, \tau^{+}_{x<z}, \tau_{\ell_1}|_{\psi_{x\geq z}}], [\varnothing, \varnothing, \{\tau^{+}_{x<z}\}, \varnothing]) \\
& && (x = z + 1 \wedge z \geq 5k \wedge y \leq z + 1) \\
\overset{\mathrm{A}}{\rightsquigarrow}\ & (\mathcal{S}_2, [\tau_{\mathsf{i}}, \tau^{+}_{x<z}, \tau^{+}_{x\geq z}], [\varnothing, \varnothing, \{\tau^{+}_{x<z}\}, \{\tau^{+}_{x\geq z}\}]) \\
& && (x \geq y \wedge x > z \geq 5k \wedge y \leq 2 \cdot z + 1) \\
\overset{\mathrm{S}}{\rightsquigarrow}\ & (\mathcal{S}_2, [\tau_{\mathsf{i}}, \tau^{+}_{x<z}, \tau^{+}_{x\geq z}, \tau_{\mathsf{err}}], [\varnothing, \varnothing, \{\tau^{+}_{x<z}\}, \{\tau^{+}_{x\geq z}\}, \varnothing]) \\
& && (x = 2 \cdot z + 1 = y \wedge z \geq 5k) \\
\overset{\mathrm{R}}{\rightsquigarrow}\ & \mathsf{unsafe}
\end{aligned}$$

Here, $5k$ abbreviates 5000 and:

$$\begin{aligned}
\psi_{x<z} &:= y \leq 2 \cdot z \wedge x{+}{+} \wedge x < z \wedge \overline{\overline{y}} \wedge \overline{\overline{z}} \qquad \psi_{x\geq z} := y \leq 2 \cdot z \wedge x{+}{+} \wedge x \geq z \wedge y{+}{+} \wedge \overline{\overline{z}} \\
\tau^{+}_{x<z} &:= \ell_1 \to \ell_1\, [\![y \leq 2 \cdot z \wedge n > 0 \wedge x' = x + n \wedge x + n \leq z \wedge \overline{\overline{y}} \wedge \overline{\overline{z}}]\!] \\
\tau^{+}_{x\geq z} &:= \ell_1 \to \ell_1\, [\![y + n - 1 \leq 2 \cdot z \wedge n > 0 \wedge x' = x + n \wedge x \geq z \wedge y' = y + n \wedge \overline{\overline{z}}]\!] \\
\mathcal{S}_1 &:= \mathcal{T} \cup \{\tau^{+}_{x<z}\} \qquad \mathcal{S}_2 := \mathcal{S}_1 \cup \{\tau^{+}_{x\geq z}\}
\end{aligned}$$

On the right, we show formulas describing the configurations that are reachable with the current trace. Every $\rightsquigarrow$-derivation starts with Init. The first two Steps add the initial transition τ_{i} and an element of $\mathsf{sip}(\tau_{\ell_1})$ to the trace. Since $x < z$ holds after applying τ_{i}, the only possible choice for the latter is $\tau_{\ell_1}|_{\psi_{x<z}}$.

As $\tau_{\ell_1}|_{\psi_{x<z}}$ is recursive, it is accelerated and replaced with $\mathsf{accel}(\tau_{\ell_1}|_{\psi_{x<z}}) = \tau^{+}_{x<z}$, which simulates n steps with $\tau_{\ell_1}|_{\psi_{x<z}}$. Moreover, $\tau^{+}_{x<z}$ is also added to the current set of blocking transitions, as we always have $\longrightarrow^{2}_{\tau} \subseteq \longrightarrow_{\tau}$ for learned transitions τ and thus adding them to the trace twice in a row is pointless.

Next, τ_{ℓ_1} is applicable again. As neither $x < z$ nor $x \geq z$ holds for all reachable configurations, we could continue with any element of $\mathsf{sip}(\tau_{\ell_1}) = \{\tau_{\ell_1}|_{\psi_{x<z}}, \tau_{\ell_1}|_{\psi_{x \geq z}}\}$. We choose $\tau_{\ell_1}|_{\psi_{x \geq z}}$, so that the recursive transition $\tau_{\ell_1}|_{\psi_{x \geq z}}$ can be accelerated to $\tau^{+}_{x \geq z}$. Then τ_{err} applies, and the proof is finished via Refute.

For our purposes, the most important property of ADCL is the following.

Theorem 7. *If $\mathcal{T} \rightsquigarrow^{*} (\mathcal{S}, \vec{\tau}, \vec{B})$ and $\vec{\tau}$ is non-empty, then $\mathsf{cond}(\vec{\tau}) \not\equiv_{\mathcal{A}} \bot$ and $\longrightarrow_{\vec{\tau}} \subseteq \longrightarrow^{+}_{\mathcal{T}}$. So if $\mathcal{T} \rightsquigarrow^{*} \mathsf{unsafe}$, then $\mathcal{T}$ is unsafe.*

The other properties of ADCL that were shown in [16] immediately carry over to our setting, too: if $\mathcal{T} \rightsquigarrow^{*} \mathsf{safe}$, then $\mathcal{T}$ is safe; if $\mathcal{T}$ is unsafe, then $\mathcal{T} \rightsquigarrow^{*} \mathsf{unsafe}$; in general, $\rightsquigarrow$ does not terminate. The proofs are analogous to [16].

4 Proving Non-Termination with ADCL-NT

From now on, we assume that the analyzed TS $\mathcal{T}$ does not contain unsafe transitions. To prove non-termination, we look for a corresponding *certificate.*

Definition 8 (Certificate of Non-Termination). *Let $\tau = \ell \to \ell[\![\ldots]\!]$. A satisfiable formula ψ* certifies non-termination *of τ, written $\psi \models^{\infty}_{\mathcal{A}} \tau$, if for any model σ of ψ, there is an infinite sequence $\ell(\sigma(\vec{x})) = \mathfrak{s}_1 \longrightarrow_{\tau} \mathfrak{s}_2 \longrightarrow_{\tau} \ldots$*

There exist many techniques for finding certificates of non-termination automatically, see Sect. 5. However, Definition 8 has several shortcomings. First, the problem of finding such certificates becomes very challenging if $\mathsf{cond}(\tau)$ contains disjunctions. Second, it is insufficient to consider a single transition when only non-singleton sequences $\vec{\tau}$ such that $\mathsf{chain}(\vec{\tau})$ is recursive admit non-terminating runs. Third, just finding a certificate ψ of non-termination for some $\vec{\tau} \in \mathcal{T}^{*}$ does not suffice for proving non-termination of $\mathcal{T}$. Additionally, a proof that the pre-image of $\longrightarrow_{\vec{\tau}|_{\psi}}$ is reachable from an initial configuration is required. All of these problems can be solved by integrating the search for certificates of non-termination into the ADCL calculus.

Definition 9 (ADCL-NT). *To prove non-termination, we extend ADCL with the rule* Nonterm *and modify* Covered *as shown below. We write $\rightsquigarrow_{\mathsf{nt}}$ for the relation defined by the (modified) rules from Definition 5 and* Nonterm.

$$\frac{\vec{\tau}^{\circlearrowleft} \text{ is recursive} \qquad \vec{\tau}^{\circlearrowleft} \sqsubset \mathsf{sip}(\mathcal{S}) \text{ or } \vec{\tau}^{\circlearrowleft} \sqsubseteq \mathsf{sip}(\mathcal{S}) \wedge |\vec{\tau}^{\circlearrowleft}| > 1}{s = (\mathcal{S}, \vec{\tau} :: \vec{\tau}^{\circlearrowleft}, \vec{B}) \rightsquigarrow_{\mathsf{nt}} \mathsf{bt}(s)} \qquad (\text{Covered})$$

$$\frac{\mathsf{chain}(\vec{\tau}^{\circlearrowleft}) = \ell \to \ell[\![\ldots]\!] \qquad \psi \models^{\infty}_{\mathcal{A}} \vec{\tau}^{\circlearrowleft} \qquad \tau = \ell \to \mathsf{err}\,[\![\psi]\!] \not\sqsubseteq \mathsf{sip}(\mathcal{S})}{(\mathcal{S}, \vec{\tau} :: \vec{\tau}^{\circlearrowleft}, \vec{B}) \rightsquigarrow_{\mathsf{nt}} (\mathcal{S} \cup \{\tau\}, \vec{\tau} :: \vec{\tau}^{\circlearrowleft}, \vec{B})} \qquad (\text{Nonterm})$$

So the idea of Nonterm is to apply a technique which searches for a certificate of non-termination to a recursive suffix of the trace. Apart from introducing Nonterm, we restricted Covered to recursive suffixes. The reason is that backtracking when the trace has a redundant, non-recursive suffix may prevent us from analyzing loops, resulting in a precision issue.

Example 10. Let $\mathcal{T} := \{\tau_{\mathsf{i}}, \tau_{\mathsf{i}}', \tau_{\ell}, \tau_{\ell'}\}$ where

$$\tau_{\mathsf{i}} := \mathsf{init} \to \ell\, [\![\top]\!] \quad \tau_{\mathsf{i}}' := \mathsf{init} \to \ell'\, [\![\top]\!] \quad \tau_{\ell} := \ell \to \ell'\, [\![\top]\!] \quad \tau_{\ell'} := \ell' \to \ell\, [\![\top]\!]$$

and $\top$ means *true*. Due to the loop $\ell \longrightarrow_{\tau_\ell} \ell' \longrightarrow_{\tau_{\ell'}} \ell$, $\mathcal{T}$ is clearly non-terminating. Without requiring that $\vec{\tau}^{\circlearrowleft}$ is recursive in Covered, $\mathcal{T}$ can be analyzed as follows:

$$\begin{aligned}\mathcal{T} &\overset{\mathrm{I}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{T}, [], [\varnothing]) \overset{\mathrm{S}}{\rightsquigarrow}{}^{2}_{\mathsf{nt}} (\mathcal{T}, [\tau_{\mathsf{i}}, \tau_{\ell}], \varnothing^3) \overset{\mathrm{C}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{T}, [\tau_{\mathsf{i}}], [\varnothing, \{\tau_{\ell}\}]) \overset{\mathrm{B}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{T}, [], [\{\tau_{\mathsf{i}}\}])\\ &\overset{\mathrm{S}}{\rightsquigarrow}{}^{2}_{\mathsf{nt}} (\mathcal{T}, [\tau_{\mathsf{i}}', \tau_{\ell'}], \{\tau_{\mathsf{i}}\} :: \varnothing^2) \overset{\mathrm{C}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{T}, [\tau_{\mathsf{i}}'], [\{\tau_{\mathsf{i}}\}, \{\tau_{\ell'}\}]) \overset{\mathrm{B}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{T}, [], [\{\tau_{\mathsf{i}}, \tau_{\mathsf{i}}'\}]) \overset{\mathrm{P}}{\rightsquigarrow}_{\mathsf{nt}} \mathsf{safe}\end{aligned}$$

The 1^{st} application of Covered is possible as $[\tau_{\mathsf{i}}, \tau_{\ell}] \sqsubseteq \tau_{\mathsf{i}}'$ and the 2^{nd} application of Covered is possible as $[\tau_{\mathsf{i}}', \tau_{\ell'}] \sqsubseteq \tau_{\mathsf{i}}$. Note that the trace never contains both τ_{ℓ} and $\tau_{\ell'}$, but both transitions are needed to prove non-termination.

Recall the shortcomings of Definition 8 mentioned above. First, due to the use of syntactic implicants, ADCL-NT reduces reasoning about arbitrary transitions to reasoning about conjunctive transitions. Second, as Nonterm considers a suffix $\vec{\tau}^{\circlearrowleft}$ of the trace, it can prove non-termination of sequences of transitions. Third, ADCL's capability to prove reachability directly carries over to our goal of proving non-termination. So in contrast to most other approaches (see Sect. 5), ADCL-NT does not have to resort to other tools or techniques for proving reachability.

We only search for a certificate of non-termination for $\vec{\tau}^{\circlearrowleft}$ if ADCL-NT established reachability of the pre-image of $\longrightarrow_{\vec{\tau}^{\circlearrowleft}}$ beforehand. Note, however, that this does not imply reachability of the pre-image of $\longrightarrow_{\ell \to \mathsf{err}\, [\![\psi]\!]}$, as ψ entails $\mathsf{cond}(\vec{\tau}^{\circlearrowleft})$, but not the other way around. Hence, we cannot directly derive non-termination of $\mathcal{T}$ when Nonterm applies. Regarding the strategy for $\rightsquigarrow_{\mathsf{nt}}$, one should try to use Nonterm once for each recursive suffix of the trace.

Example 11. Reconsider Example 1. Up to (excluding) the second-last step, the derivation from Example 6 remains unchanged. Then we get

$$\begin{aligned}&(\mathcal{S}_2, [\tau_{\mathsf{i}}, \tau^{+}_{x<z}, \tau^{+}_{x\geq z}], [\ldots]) && (x \geq y \wedge x > 5k)\\ \overset{\mathrm{S}}{\rightsquigarrow}{}^{4}_{\mathsf{nt}}\ &(\mathcal{S}_2, [\tau_{\mathsf{i}}, \tau^{+}_{x<z}, \tau^{+}_{x\geq z}, \tau_{\ell_1 \to \ell_2}, \tau^{=}_{\ell_2}, \tau^{\neq}_{\ell_2}|\psi_{x>y}, \tau^{\neq}_{\ell_2}|\psi_{x<y}], [\ldots]) && (1 \equiv_2 y = x > 10k)\\ \overset{\mathrm{N}}{\rightsquigarrow}_{\mathsf{nt}}\ &(\mathcal{S}_3, [\tau_{\mathsf{i}}, \tau^{+}_{x<z}, \tau^{+}_{x\geq z}, \tau_{\ell_1 \to \ell_2}, \tau^{=}_{\ell_2}, \tau^{\neq}_{\ell_2}|\psi_{x>y}, \tau^{\neq}_{\ell_2}|\psi_{x<y}], [\ldots]) && (1 \equiv_2 y = x > 10k)\\ \overset{\mathrm{S}}{\rightsquigarrow}_{\mathsf{nt}}\ &(\mathcal{S}_3, [\tau_{\mathsf{i}}, \tau^{+}_{x<z}, \tau^{+}_{x\geq z}, \tau_{\ell_1 \to \ell_2}, \tau^{=}_{\ell_2}, \tau^{\neq}_{\ell_2}|\psi_{x>y}, \tau^{\neq}_{\ell_2}|\psi_{x<y}, \tau_{\mathsf{err}}], [\ldots]) \overset{\mathrm{R}}{\rightsquigarrow}_{\mathsf{nt}} \mathsf{unsafe}\end{aligned}$$

$$\begin{aligned}\text{where}\quad &\psi_{x>y} := x > 0 \wedge y > 0 \wedge x' = y \wedge x > y \wedge y' = x && \tau_{\mathsf{err}} := \ell_2 \to \mathsf{err}\, [\![x = y > 1]\!]\\ &\psi_{x<y} := x > 0 \wedge y > 0 \wedge x' = y \wedge x < y \wedge \overset{=}{y} && \mathcal{S}_3 := \mathcal{S}_2 \cup \{\tau_{\mathsf{err}}\}\end{aligned}$$

The formulas on the right describe the values of x and y that are reachable with the current trace, where $1 \equiv_2 y$ means that y is odd. After the first Step with

$\tau_{\ell_1 \to \ell_2}$, just $\tau_{\ell_2}^{=}$ can be used, as $\mathsf{cond}(\tau_{\ell_1 \to \ell_2})$ implies $x' = y'$. While $\tau_{\ell_2}^{=}$ is recursive, Accelerate cannot be applied next, as $\longrightarrow_{\tau_{\ell_2}^{=}} = \longrightarrow_{\tau_{\ell_2}^{=}}^{+}$, so the learned transition would be redundant. Thus, we continue with $\tau_{\ell_2}^{\neq}$, projected to $x > y$ (as $\mathsf{cond}(\tau_{\ell_2}^{=})$ implies $x' = y' + 1$). Again, all transitions that could be learned are redundant, so Accelerate does not apply. We next use $\tau_{\ell_2}^{\neq}$ projected to $x < y$, as the previous Step swapped x and y. As the suffix $[\tau_{\ell_2}^{=}, \tau_{\ell_2}^{\neq}|_{\psi_{x>y}}, \tau_{\ell_2}^{\neq}|_{\psi_{x<y}}]$ of the trace does not terminate (see Example 1), Nonterm applies. So we learn the transition τ_{err}, which is added to the trace to finish the proof, afterwards.

Theorem 12. *If $\mathcal{T} \rightsquigarrow_{\mathsf{nt}}^{*} \mathsf{unsafe}$, then $\mathcal{T}$ does not terminate.*

While Theorem 12 establishes the soundness of our approach, we now investigate completeness. In contrast to ADCL for safety (Sect. 3), ADCL-NT is not refutationally complete, but the proof is non-trivial. So in the following, we show that there are non-terminating TSs $\mathcal{T}$ where $\mathcal{T} \not\rightsquigarrow_{\mathsf{nt}}^{*} \mathsf{unsafe}$. To prove incompleteness, we adapt the construction from the proof that ADCL does not terminate [16, Thm. 18]. There, states $(\mathcal{S}, \vec{\tau}, \vec{B})$ were extended by a component $\mathcal{L}$ that maps every element of $\mathsf{sip}(\mathcal{S})$ to a regular language over $\mathsf{sip}(\mathcal{T})$. However, the proof of [16, Thm. 18] just required reasoning about finite (prefixes of infinite) runs, but we have to reason about infinite runs. So in our setting $\mathcal{L}$ maps each element τ of $\mathsf{sip}(\mathcal{S})$ to a regular or an ω-regular language over $\mathsf{sip}(\mathcal{T})$, i.e., $\mathcal{L}(\tau) \subseteq \mathsf{sip}(\mathcal{T})^{*}$ or $\mathcal{L}(\tau) \subseteq \mathsf{sip}(\mathcal{T})^{\omega}$. We lift $\mathcal{L}$ from $\mathsf{sip}(\mathcal{S})$ to sequences of transitions as follows.

$$\mathcal{L}(\varepsilon) := \varepsilon \qquad\qquad \mathcal{L}(\vec{\tau} :: \tau) := \mathcal{L}(\vec{\tau}) :: \mathcal{L}(\tau) \quad \text{if} \quad \mathcal{L}(\tau) \subseteq \mathsf{sip}(\tau)^{*}$$

Here, "::" denotes language concatenation (i.e., $\mathcal{L}_1 :: \mathcal{L}_2 = \{\tau_1 :: \tau_2 \mid \tau_1 \in \mathcal{L}_1, \tau_2 \in \mathcal{L}_2\}$) and we only consider sequences where $\mathcal{L}(\tau)$ is regular (not ω-regular) to ensure that $\mathcal{L}$ is well defined. So while we lift other notations to sequences of transitions via chaining, $\mathcal{L}(\vec{\tau})$ does *not* stand for $\mathcal{L}(\mathsf{chain}(\vec{\tau}))$.

Definition 13 (ADCL-NT with Regular Languages). *We extend states by a fourth component $\mathcal{L}$, and adapt* Init, Accelerate, *and* Nonterm *as follows:*

$$\frac{\mathcal{L}(\tau) = \{\tau\} \textit{ for all } \tau \in \mathsf{sip}(\mathcal{T})}{\mathcal{T} \rightsquigarrow_{\mathsf{nt}} (\mathcal{T}, [], [\varnothing], \mathcal{L})} \quad (\textsc{Init})$$

$$\frac{\vec{\tau}^{\circlearrowleft} \textit{ is recursive} \qquad |\vec{\tau}^{\circlearrowleft}| = |\vec{B}^{\circlearrowleft}| \qquad \mathsf{accel}(\vec{\tau}^{\circlearrowleft}) = \tau \not\sqsubseteq \mathsf{sip}(\mathcal{S})}{(\mathcal{S}, \vec{\tau} :: \vec{\tau}^{\circlearrowleft}, \vec{B} :: \vec{B}^{\circlearrowleft}, \mathcal{L}) \rightsquigarrow_{\mathsf{nt}} (\mathcal{S} \cup \{\tau\}, \vec{\tau} :: \tau, \vec{B} :: \{\tau\}, \mathcal{L} \uplus (\tau \mapsto \mathcal{L}(\vec{\tau}^{\circlearrowleft})^{+}))} \quad (\textsc{Accelerate})$$

$$\frac{\mathsf{chain}(\vec{\tau}^{\circlearrowleft}) = \ell \to \ell \, [\![\ldots]\!] \quad \psi \models_{\mathcal{A}}^{\infty} \vec{\tau}^{\circlearrowleft} \quad \tau = \ell \to \mathsf{err} \, [\![\psi]\!] \not\sqsubseteq \mathsf{sip}(\mathcal{S})}{(\mathcal{S}, \vec{\tau} :: \vec{\tau}^{\circlearrowleft}, \vec{B}, \mathcal{L}) \rightsquigarrow_{\mathsf{nt}} (\mathcal{S} \cup \{\tau\}, \vec{\tau} :: \vec{\tau}^{\circlearrowleft}, \vec{B}, \mathcal{L} \uplus (\tau \mapsto \mathcal{L}(\vec{\tau}^{\circlearrowleft})^{\omega}))} \quad (\textsc{Nonterm})$$

All other rules from Definition 5 leave the last component of the state unchanged.

Here, $\mathcal{L}(\pi)^{+} := \bigcup_{n \in \mathbb{N}_{\geq 1}} \mathcal{L}(\pi)^{n}$, and $\mathcal{L}(\pi)^{\omega}$ is the ω-regular language consisting of all words that result from concatenating infinitely many elements of $\mathcal{L}(\pi) \setminus \{\varepsilon\}$.

In Accelerate and Nonterm, $\mathsf{chain}(\vec{\tau}^{\circlearrowleft})$ is recursive. Thus, $\vec{\tau}^{\circlearrowleft}$ does not contain unsafe transitions. Hence, $\mathcal{L}(\vec{\tau}^{\circlearrowleft})$ and thus also $\mathcal{L}(\vec{\tau}^{\circlearrowleft})^{+}$ are well defined and regular, and $\mathcal{L}(\vec{\tau}^{\circlearrowleft})^{\omega}$ is ω-regular. Moreover, the use of "$\uplus$" is justified by the condition $\tau \not\sqsubseteq \mathsf{sip}(\mathcal{S})$. The next lemma states two crucial properties about $\mathcal{L}$.

Lemma 14. *Assume* $\mathcal{T} \rightsquigarrow_{\mathsf{nt}}^{*} (\mathcal{S}, \vec{\tau}, \vec{B}, \mathcal{L})$ *and let* $\tau = (\ell \to \ell' \, [\![\psi]\!]) \in \mathsf{sip}(\mathcal{S})$.

- *If* $\mathcal{L}(\tau) \subseteq \mathsf{sip}(\mathcal{T})^{*}$*, then* $\longrightarrow_{\tau} = \bigcup_{\vec{\tau} \in \mathcal{L}(\tau)} \longrightarrow_{\vec{\tau}}$.
- *If* $\mathcal{L}(\tau) \subseteq \mathsf{sip}(\mathcal{T})^{\omega}$*, then for every model* σ *of* ψ*, there is an infinite sequence* $\ell(\sigma(\vec{x})) = \mathfrak{s}_1 \longrightarrow_{\tau_1} \mathfrak{s}_2 \longrightarrow_{\tau_2} \ldots$ *where* $[\tau_1, \tau_2, \ldots] \in \mathcal{L}(\tau)$.

Based on this lemma, we can prove that our extension of $\rightsquigarrow_{\mathsf{nt}}$ from Definition 13 is not refutationally complete. Then refutational incompleteness of ADCL-NT as introduced in Definition 9 follows immediately. The reason is that $\mathcal{L}$ is only used in the premise of Init in Definition 13, but there the requirement "$\mathcal{L}(\tau) = \{\tau\}$ for all $\tau \in \mathsf{sip}(\mathcal{T})$" is trivially satisfiable by choosing $\mathcal{L}$ accordingly.

Theorem 15. *There is a non-terminating TS* $\mathcal{T}$ *such that* $\mathcal{T} \not\rightsquigarrow_{\mathsf{nt}}^{*} \mathsf{unsafe}$.

Proof (Sketch). As in the proof of [16, Thm. 18], for any (original or learned) transition τ such that $\mathcal{L}(\tau)$ is regular, $\mathcal{L}(\tau)$ contains at most one square-free word (i.e., a word without a non-empty infix $w :: w$). Thus, if $\mathcal{L}(\tau)$ is ω-regular, then $\mathcal{L}(\tau)$ does not contain an infinite square-free word. Moreover, as in the proof of [16, Thm. 18], one can construct a TS $\mathcal{T}$ that admits a single infinite run $\vec{\tau}$, and this infinite run is square-free. Thus, there is no transition τ such that $\mathcal{L}(\tau)$ contains a suffix of $\vec{\tau}$, i.e., no $\rightsquigarrow_{\mathsf{nt}}$-derivation starting with $\mathcal{T}$ corresponds to $\vec{\tau}$. Hence, by Lemma 14, assuming $\mathcal{T} \rightsquigarrow_{\mathsf{nt}}^{*} \mathsf{unsafe}$ results in a contradiction. □

Since ADCL can prove unsafety as well as safety, it is natural to ask if there is a dual to ADCL-NT that can prove termination. The most obvious approach would be the following: Whenever the trace has a recursive suffix $\vec{\tau}^{\circlearrowleft}$, then termination of $\vec{\tau}^{\circlearrowleft}$ needs to be proven before the next $\rightsquigarrow$-step. The following example shows that this is not enough to ensure that $\mathcal{T} \rightsquigarrow_{\mathsf{nt}}^{+} \mathsf{safe}$ implies termination of $\mathcal{T}$.

Example 16. Let $\mathcal{T} := \{\tau_{\mathsf{i}} = \mathsf{init} \to \ell \, [\![\psi_{\mathsf{i}}]\!]\} \cup \{\tau_m = \ell \to \ell \, [\![\psi_m]\!] \mid 0 \leq m \leq 2\}$ and

$$\psi_{\mathsf{i}} := x' = 0 \quad \psi_0 := x = 0 \wedge x' = 1 \quad \psi_1 := x = 1 \wedge x' = 2 \quad \psi_2 := x = 2 \wedge x' = 1.$$

As we have $\ell(1) \longrightarrow_{\tau_1} \ell(2) \longrightarrow_{\tau_2} \ell(1)$, $\mathcal{T}$ is clearly non-terminating. We get:

$$\begin{aligned} \mathcal{T} & \overset{\mathrm{I}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{T}, [], [\varnothing]) \overset{\mathrm{S}}{\rightsquigarrow}_{\mathsf{nt}}^{3} (\mathcal{T}, [\tau_{\mathsf{i}}, \tau_0, \tau_1], \varnothing^4) \overset{\mathrm{A}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{S}_1, [\tau_{\mathsf{i}}, \tau_{01}], \varnothing^2 :: \{\tau_{01}\}) \\ & \overset{\mathrm{S}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{S}_1, [\tau_{\mathsf{i}}, \tau_{01}, \tau_2], \varnothing^2 :: \{\tau_{01}\} :: \varnothing) \overset{\mathrm{A}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{S}_2, [\tau_{\mathsf{i}}, \tau_{012}], \varnothing^2 :: \{\tau_{01}, \tau_{012}\}) \\ & \overset{\mathrm{S}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{S}_2, [\tau_{\mathsf{i}}, \tau_{012}, \tau_1], \varnothing^2 :: \{\tau_{01}, \tau_{012}\} :: \varnothing) \overset{\mathrm{C}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{S}_2, [\tau_{\mathsf{i}}, \tau_{012}], \varnothing^2 :: \{\tau_{01}, \tau_{012}, \tau_1\}) \\ & \overset{\mathrm{B}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{S}_2, [\tau_{\mathsf{i}}], \varnothing :: \{\tau_{012}\}) \rightsquigarrow_{\mathsf{nt}}^{*} (\mathcal{S}_2, [\tau_{\mathsf{i}}], \varnothing :: \{\tau_{012}, \tau_0, \tau_{01}\}) \overset{\mathrm{B}}{\rightsquigarrow}_{\mathsf{nt}} (\mathcal{S}_2, [], [\{\tau_{\mathsf{i}}\}]) \overset{\mathrm{P}}{\rightsquigarrow}_{\mathsf{nt}} \mathsf{safe} \end{aligned}$$

After three Steps, we accelerate the recursive suffix $[\tau_0, \tau_1]$ of the trace, resulting in $\tau_{01} = \ell \to \ell \, [\![x = 0 \wedge x' = 2]\!]$ and $\mathcal{S}_1 = \mathcal{T} \cup \{\tau_{01}\}$. After one more step, $[\tau_{01}, \tau_2]$

is accelerated to $\tau_{012} = \ell \to \ell\,[\![x = 0 \land x' = 1]\!]$ and we get $\mathcal{S}_2 = \mathcal{S}_1 \cup \{\tau_{012}\}$. After the next step, $[\tau_{012}, \tau_1]$ is redundant w.r.t. τ_{01}, so Covered applies. Then we Backtrack, as no other transitions are active. The next Steps also yield states that allow for backtracking (as their traces have the redundant suffixes $[\tau_0, \tau_1]$ and $[\tau_{01}, \tau_2]$), so we can finally apply Backtrack again and finish with Prove.

Note that whenever the trace has a recursive suffix, then it leads from $\ell(i)$ to $\ell(j)$ where $i \neq j$, i.e., each such suffix is trivially terminating. In particular, the cycle $\ell(1) \longrightarrow_{\tau_1} \ell(2) \longrightarrow_{\tau_2} \ell(1)$ is not apparent in any of the states.

This example reveals a fundamental problem when adapting ADCL for proving termination: ADCL ensures that all reachable *configurations* are covered, which is crucial for proving safety, but there are no such guarantees for all *runs*. Therefore, we think that adapting ADCL for proving termination requires major changes.

5 Related Work and Experiments

We presented ADCL-NT, a variant of ADCL for proving non-termination. The key insight is that tightly integrating techniques to detect non-terminating transitions into ADCL allows for handling classes of TSs that are challenging for other techniques. In particular, ADCL-NT can find non-terminating executions involving disjunctive transitions or complex patterns of transitions. Moreover, it tightly couples the search for non-terminating configurations and the proof of their reachability, whereas other approaches usually separate these two steps.

Related Work. There are many techniques to find certificates of non-termination [2,14,15,22,23,25]. We could use any of them (they are black boxes for ADCL-NT).

Most non-termination techniques for TSs first search for non-terminating configurations, and then prove their reachability [5,6,9,22], or they extract and analyze *lassos* [23]. In contrast, ADCL-NT tightly integrates the search for non-terminating configurations and reachability analysis.

Earlier versions of our tool LoAT [12,15] also interleaved both steps using a technique akin to the state elimination method to transform finite automata to regular expressions. This technique cannot handle disjunctions, and it is incomplete for reachability. Hence, LoAT is now solely based on ADCL-NT.

Implementation. So far, our implementation in our tool LoAT is restricted to integer arithmetic. It uses the technique from [15] for acceleration and finding certificates of non-termination, the SMT solvers Z3 [26] and Yices [11], the recurrence solver PURRS [1], and libFAUDES [24] to implement the automata-based redundancy check from [16].

Experiments. To evaluate our implementation in LoAT, we used the 1222 *Integer Transition Systems* (ITSs) and the 335 C *Integer Programs* from the *Termination Problems Database* [28] used in *TermComp* [21]. The C programs are small, hand-crafted examples that often require complex proofs. The ITSs are significantly larger, as they were obtained from automatic transformations of C or Java programs. Moreover, they contain a lot of "noise", e.g., branches where termination is trivial or variables that are irrelevant for (non-)termination. Thus, they are well suited to test the scalability and robustness of the tools.

We compared our implementation (LoAT ADCL) with other leading termination analyzers: iRankFinder [2,9], T2 [6], Ultimate [8], VeryMax [3,22], and the previous version of LoAT [15] (LoAT '22). For T2, VeryMax, and Ultimate, we took the versions of their last *TermComp* participations (2015, 2019, and 2022). For iRankFinder, we used the configuration from the evaluation of [15], which is tailored towards proving non-termination. We excluded AProVE [20], as it cannot prove non-termination of ITSs, and it uses LoAT and T2 as backends when analyzing C programs. Moreover, we excluded Ultimate from the evaluation on ITSs, as it cannot parse them. All experiments were run on StarExec [27] with 300 s wallclock timeout, 1200 s CPU timeout, and 128 GB memory limit per example.

	No		Yes	Runtime overall			Runtime No	
	solved	unique	solved	average	median	timeouts	average	median
LoAT ADCL	521	9	0	48.6 s	0.1 s	183	2.9 s	0.1 s
LoAT '22	494	2	0	7.4 s	0.1 s	0	6.2 s	0.1 s
T2	442	3	615	17.2 s	0.6 s	45	7.4 s	0.6 s
VeryMax	421	6	631	28.3 s	0.5 s	30	30.5 s	14.5 s
iRankFinder	409	0	642	32.0 s	2.0 s	93	12.3 s	1.7 s

The table above shows the results for ITSs, where the column "unique" contains the number of examples that could be solved by the respective tool, but no others. It shows that LoAT ADCL is the most powerful tool for proving non-termination of ITSs. The main reasons for the improvement are that LoAT ADCL builds upon a complete technique for proving reachability (in contrast to, e.g., LoAT '22), and the close integration of non-termination techniques into a technique for proving reachability, whereas most competing tools separate these steps from each other.

If we only consider the examples where non-termination is proven, LoAT ADCL is also the fastest tool. If we consider all examples, then the *average* runtime of LoAT ADCL is significantly slower. This is not surprising, as ADCL-NT does not terminate in general. So while it is very fast in most cases (as witnessed by the very fast *median* runtime), it times out more often than the other tools.

For C integer programs, the best tools are very close (VeryMax: 103×No, LoAT ADCL: 102×No, Ultimate: 100×No). Regarding runtimes, the situation is analogous to ITSs. See [18] for detailed results, more information about our evaluation, and a pre-compiled binary. LoAT is open-source and available on GitHub [17].

References

1. Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E.: PURRS: towards computer algebra support for fully automatic worst-case complexity analysis. CoRR abs/cs/0512056 (2005). https://arxiv.org/abs/cs/0512056
2. Ben-Amram, A.M., Doménech, J.J., Genaim, S.: Multiphase-linear ranking functions and their relation to recurrent sets. In: Chang, B.-Y.E. (ed.) SAS 2019. LNCS, vol. 11822, pp. 459–480. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32304-2_22
3. Borralleras, C., Brockschmidt, M., Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving termination through conditional termination. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 99–117. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_6
4. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 337–351. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_29
5. Brockschmidt, M., Ströder, T., Otto, C., Giesl, J.: Automated detection of non-termination and NullPointerExceptions for Java Bytecode. In: Beckert, B., Damiani, F., Gurov, D. (eds.) Formal Verification of Object-Oriented Software. FoVeOOS 2011. LNCS, vol. 7421, pp. 123–141. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31762-0_9
6. Brockschmidt, M., Cook, B., Ishtiaq, S., Khlaaf, H., Piterman, N.: T2: temporal property verification. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 387–393. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_22
7. CHC Competition. https://chc-comp.github.io
8. Chen, Y., et al.: Advanced automata-based algorithms for program termination checking. In: Foster, J.S., Grossman, D. (eds.) PLDI 2018, pp. 135–150 (2018). https://doi.org/10.1145/3192366.3192405
9. Doménech, J.J., Genaim, S.: iRankFinder. In: Lucas, S. (ed.) WST 2018, p. 83 (2018). https://wst2018.webs.upv.es/wst2018proceedings.pdf
10. Doménech, J.J., Gallagher, J.P., Genaim, S.: Control-flow refinement by partial evaluation, and its application to termination and cost analysis. Theory Pract. Log. Program. **19**(5–6), 990–1005 (2019). https://doi.org/10.1017/S1471068419000310
11. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
12. Frohn, F., Giesl, J.: Proving non-termination via loop acceleration. In: Barrett, C.W., Yang, J. (eds.) FMCAD 2019, pp. 221–230 (2019). https://doi.org/10.23919/FMCAD.2019.8894271
13. Frohn, F.: A calculus for modular loop acceleration. In: Biere, A., Parker, D. (eds.) TACAS 2020. LNCS, vol. 12078, pp. 58–76. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_4
14. Frohn, F., Fuhs, C.: A calculus for modular loop acceleration and non-termination proofs. Int. J. Softw. Tools Technol. Transf. **24**(5), 691–715 (2022). https://doi.org/10.1007/s10009-022-00670-2
15. Frohn, F., Giesl, J.: Proving non-termination and lower runtime bounds with LoAT (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning. IJCAR 2022. LNCS, vol. 13385, pp. 712–722. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_41

16. Frohn, F., Giesl, J.: ADCL: Acceleration Driven Clause Learning for constrained Horn clauses. CoRR abs/2303.01827 (2023). https://arxiv.org/abs/2303.01827
17. Frohn, F.: LoAT on GitHub (2023). https://github.com/LoAT-developers/LoAT
18. Frohn, F., Giesl, J.: Empirical evaluation of "Proving non-termination by Acceleration Driven Clause Learning" (2023). https://loat-developers.github.io/adcl-nonterm-eval
19. Frohn, F., Giesl, J.: Proving non-termination by Acceleration Driven Clause Learning. CoRR abs/2304.10166 (2023). https://arxiv.org/abs/2304.10166
20. Giesl, J., et al.: Analyzing program termination and complexity automatically with AProVE. J. Autom. Reason. **58**(1), 3–31 (2017). https://doi.org/10.1007/s10817-016-9388-y
21. Giesl, J., Rubio, A., Sternagel, C., Waldmann, J., Yamada, A.: The termination and complexity competition. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 156–166. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_10
22. Larraz, D., Nimkar, K., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving non-termination using Max-SMT. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 779–796. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_52
23. Leike, J., Heizmann, M.: Geometric nontermination arguments. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 266–283. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_16
24. libFAUDES Library. https://fgdes.tf.fau.de/faudes/index.html
25. Nishida, N., Winkler, S.: Loop detection by logically constrained term rewriting. In: Piskac, R., Rümmer, P. (eds.) VSTTE 2018. LNCS, vol. 11294, pp. 309–321. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03592-1_18
26. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
27. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: a cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 367–373. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_28
28. Termination Problems Data Base (TPDB). https://termination-portal.org/wiki/TPDB

COOL 2 – A Generic Reasoner for Modal Fixpoint Logics (System Description)

Oliver Görlitz[1], Daniel Hausmann[2], Merlin Humml[1](✉), Dirk Pattinson[3], Simon Prucker[1], and Lutz Schröder[1]

[1] Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany
merlin.humml@fau.de
[2] Gothenburg University, Gothenburg, Sweden
[3] Australian National University, Canberra, Australia

Abstract. There is a wide range of modal logics whose semantics goes beyond relational structures, and instead involves, e.g., probabilities, multi-player games, weights, or neighbourhood structures. Coalgebraic logic serves as a unifying semantic and algorithmic framework for such logics. It provides uniform reasoning algorithms that are easily instantiated to particular, concretely given logics. The *COOL 2* reasoner provides an implementation of such generic algorithms for coalgebraic modal fixpoint logics. As concrete instances, we obtain in particular reasoners for the aconjunctive and alternation-free fragments of the graded μ-calculus and the alternating-time μ-calculus. We evaluate the tool on standard benchmark sets for fixpoint-free graded modal logic and alternating-time temporal logic (ATL), as well as on a dedicated set of benchmarks for the graded μ-calculus.

1 Introduction

Modal and temporal logics are established tools in the specification and verification of systems. While many such logics are interpreted over relational transition systems, the semantics of quite a number of important logics goes beyond the relational setup, involving, for instance, probabilities [20,30], concurrent games as in alternating-time logics [1,36], monotone neighbourhoods structures as in game logic [34] and concurrent dynamic logic [37], or integer transition weights as in the multigraph semantics [5] of the graded μ-calculus [25]. *Coalgebraic logic* [4] provides a uniform semantic and algorithmic framework for these logics, based on the paradigm of *universal coalgebra* [38]. It provides reasoning algorithms of optimal complexity at various levels of expressiveness, up to the coalgebraic

D. Hausmann—Supported by the ERC Consolidator grant D-SynMA (No. 772459)
M. Humml—Supported by Deutsche Forschungsgemeinschaft (DFG) as part of the Research Training Group 2475 (grant number 393541319/GRK2475/1-2019) and the project 'RAND' (grant number 377333057).
L. Schröder—Supported by Deutsche Forschungsgemeinschaft (DFG) under project no. 419850228.

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 234–247, 2023.
https://doi.org/10.1007/978-3-031-38499-8_14

μ-calculus [3,21–23]. These algorithms are parametric in the transition type of systems (weighted, probabilistic, game-based etc.) as well as in suitable choices of modalities specific to the given system type. Their instantiation to specific logics requires providing either a set of next-step modal tableau rules satisfying a suitable completeness criterion [41] or, more generally, a plug-in algorithm that determines satisfiability for an extremely simple *one-step logic* that describes the interaction between modalities, and consists of (conjunctions of) modal operators applied to variables only [29].

The *COalgebraic Ontology Logic solver (COOL)* provides reasoning support for coalgebraic logics based on these generic algorithms. The first version of the tool [15] provided reasoning support for fixpoint-free coalgebraic hybrid logic with global assumptions, using a global caching principle [13]. In the present paper, we present *COOL 2*, which provides reasoning support for coalgebraic fixpoint logics, specifically for both the aconjunctive fragment and the alternation-free fragment of the coalgebraic μ-calculus. By instantiation, we obtain in particular the first implemented reasoners for the graded μ-calculus [26] (for which a set of coalgebraic modal tableau rules has been described in the literature [41]; however, this rule set has later turned out to be incomplete, cf. Remark 2.3) and the alternating-time μ-calculus [1]. We describe the structure of the tool including implementational details, and present evaluation results, focusing on the graded μ-calculus and alternating-time temporal logic (ATL). Additional details on the evaluation can be found in the full version [17].

Related Work: We have already mentioned work in coalgebraic logic on which COOL is based [3,13,21–23,41]. COOL is conceptually a successor of the *Coalgebraic Logic Satisfiability Solver (CoLoSS)* [2] but does not share any of its code. CoLoSS implements fixpoint-free logics, and is entirely unoptimised. The first version of COOL [15] has been evaluated on fixpoint-free next-step logics.

COOL does cover also various relational modal logics, for which there are numerous specialised reasoners, including highly optimised description logic reasoners such as FaCT++ [44], Pellet [42], RACER [18], and HermiT [12]. As these systems do not support fixpoint logics, a comparison would be of limited value. In previous work, COOL has been evaluated on various relational fixpoint logics, and has been shown to perform favourably on Computation Tree Logic [23] (in comparison to reasoners featured in a previous systematic evaluation [14]), as well as on the aconjunctive fragment of the modal μ-calculus [22] (in comparison to MLSolver [11]). A reasoner for (next-step) graded modal logic has been evaluated against various description logic reasoners [43], using however the above-mentioned incomplete set of modal tableau rules.

For the same reasons, we refrain from evaluating COOL 2 against reasoners for coalition logic, i.e. the fixpoint-free fragment of the alternating-time μ-calculus, such as CLProver [32]. The only implemented reasoner for any fragment of the alternating-time μ-calculus that does include fixpoints still appears to be the tableau reasoner TATL for alternating-time temporal logic [6,7]. TATL has been compared to COOL on random formulas in previous work [23].

2 Satisfiability in the Coalgebraic μ-Calculus

COOL 2 is a satisfiability checker for the coalgebraic μ-calculus [3], that is, for the extension of coalgebraic modal logic with extremal fixpoint operators. Formulas of this logic are interpreted over coalgebras, where the semantics of modal operators is defined by means of so-called *predicate liftings* [41]; we recapitulate examples of system types and modalities subsumed by this paradigm in Example 2.1.

Syntax: Formulas are built relative to a set Var of fixpoint variables and a *modal similarity type* Λ, that is, a set of modal operators with assigned finite arities that is closed under duals, with $\overline{\heartsuit} \in \Lambda$ denoting the dual of $\heartsuit \in \Lambda$. Formulas $\psi, \phi, \ldots$ of the *coalgebraic μ-calculus* over Λ are given by the grammar

$$\psi, \phi := \bot \mid \top \mid \psi \wedge \phi \mid \psi \vee \phi \mid \heartsuit(\psi_1, \ldots, \psi_n) \mid X \mid \mu X.\, \psi \mid \nu X.\, \psi,$$

where $\heartsuit \in \Lambda$ has arity n and $X \in \mathsf{Var}$. A formula χ is *aconjunctive* if for every conjunction $\psi \wedge \phi$ that is a subformula of χ, at most one of the formulas ψ and ϕ contains a free fixpoint variable X that is bound by a least fixpoint operator μX. While the logic does not contain negation as an explicit operator, full negation can be defined as usual; e.g. we have $\neg\heartsuit\psi = \overline{\heartsuit}\neg\psi$ and $\neg\mu X.\, \psi = \nu X.\, \neg\psi[\neg X/X]$, using $\neg\neg X = X$.

Both the theoretical satisfiability checking algorithm and its implementation in COOL 2 operate on the *Fischer-Ladner closure* [21,24,27] of the target formula. The *alternation depth* (e.g. [21,29,33]) of a formula is the maximum depth of dependent alternating nestings of least and greatest fixpoints within the formula. Formulas with alternation depth 1 are *alternation-free*.

Semantics: Formulas are interpreted over F-coalgebras, that is, structures

$$(C, \xi : C \to FC),$$

where $F\colon \mathsf{Set} \to \mathsf{Set}$ is a functor determining the branching type of the systems at hand; thus $\xi(x) \in FC$ encodes the transitions from $x \in C$, structured according to F. Modalities $\heartsuit \in \Lambda$ of arity n are interpreted as *predicate liftings*, that is, families of maps $[\![\heartsuit]\!]_U\colon (2^U)^n \to 2^{FU}$ (for $U \in \mathsf{Set}$) that assign predicates on FU to n-tuples of predicates on U, subject to a *naturality* condition [35,40]. On a coalgebra (C, ξ), the semantics of formulas is defined inductively in the usual way for the propositional operators and fixpoints, and by $[\![\heartsuit(\psi_1, \ldots, \psi_n)]\!] = \xi^{-1}[[\![\heartsuit]\!]_C([\![\psi_1]\!], \ldots, [\![\psi_n]\!])]$ for modalities.

A closed formula ψ is *satisfiable* if there is a coalgebra (C, ξ) and a state $x \in C$ such that $x \in [\![\psi]\!]$. A formula ψ is *valid* if $\neg\psi$ is not satisfiable.

Example 2.1.(1) The standard *modal μ-calculus* [24] is obtained using the functor $F = \mathcal{P}(A) \times \mathcal{P}$, where A is a fixed set of atoms, the similarity type $\Lambda = \{\Diamond, \Box, a, \neg a \mid a \in A\}$, and predicate liftings

$$[\![\Diamond]\!]_C(B) = \{(A, Z) \in 2^A \times 2^C \mid Z \cap B \neq \emptyset\} \qquad [\![a]\!]_C = \{(A, Z) \in 2^A \times 2^C \mid a \in A\}$$

$$[\![\Box]\!]_C(B) = \{(A, Z) \in 2^A \times 2^C \mid Z \subseteq B\} \qquad [\![\neg a]\!]_C = \{(A, Z) \in 2^A \times 2^C \mid a \notin A\}$$

The expressive power of the modal μ-calculus is demonstrated by the formulas

$$\mu X.\, \nu Y.\, (p \wedge \Diamond Y) \vee \Diamond X \qquad\qquad \nu X.\, \mu Y.\, (p \wedge \Diamond X) \vee \Diamond Y.$$

The former is a co-Büchi formula expressing the existence of a path on which p holds forever, from some point on; the latter formula expresses the Büchi property that there is a path on which the atom p is satisfied infinitely often.

(2) The *graded μ-calculus* [26] allows expressing quantitative properties with the help of modal operators $\langle n\rangle$ and $[n]$, $n \in \mathbb{N}$; formulas $\langle n\rangle\psi$ and $[n]\psi$ then have the intuitive meaning that 'there are more than n successor states that satisfy ψ', and 'all but at most n successor states satisfy ψ', respectively. Its coalgebraic interpretation is based on *multigraphs*, which are coalgebras for the multiset functor [5]. A graded variant of the above Büchi property is specified, e.g., by the formula $\nu X.\, \mu Y.\, (p \wedge \langle n\rangle X) \vee \langle n\rangle Y$, which expresses the existence of an infinite $n+1$-ary tree such that the atom p is satisfied infinitely often on every path in the tree.

(3) The *alternating-time μ-calculus* (AMC) [39] extends coalition logic [36] with fixpoints and (modulo syntax) supports modalities $\langle D\rangle$ and $[D]$, where $D \subseteq N$ is a coalition formed by agents from the set $N = \{1, \dots, n\}$ for some fixed $n \in \mathbb{N}$; formulas $\langle D\rangle\psi$ and $[D]\psi$ then state that 'coalition D has a joint strategy to enforce ψ' and that 'coalition D cannot prevent ψ', respectively. For instance, the formula $\nu X.\, \mu Y.\, \nu Z.\, (p \wedge \langle D\rangle X) \vee (q \wedge \langle D\rangle Y) \vee (\neg q \wedge \langle D\rangle Z)$ expresses that coalition D has a joint multi-step strategy that guarantees that p is visited infinitely often whenever q is visited infinitely often.

Satisfiability Checking: We proceed to recall the satisfiability checking algorithm for the coalgebraic μ-calculus that forms the basis of the implementation within COOL 2. This algorithm adapts the automata-based approach to satisfiability checking for the standard μ-calculus, and generalises the treatment of modal steps by parametrizing over a solver for the *one-step satisfiability* problem of the logic, which concerns satisfiability of formulae with exactly one layer of next-step modalities [21]. It thus avoids the necessity of tractable sets of tableaux rules for modal operators. Under mild assumptions on the complexity of the one-step satisfiability problem of the base logic at hand ('*tractability*'), the algorithm witnesses a, typically optimal, upper bound ExpTime for the complexity of the satisfiability problem; unlike a previous algorithm [4], the algorithm thus has optimal runtime also in cases where no tractable sets of modal tableaux rules are known, such as the graded (or, more generally, Presburger) μ-calculus (further cases of this kind include the probabilistic μ-calculus with polynomial inequalities [21] and the unrestricted form of the *alternating-time μ-calculus with disjunctive explicit strategies* [16]).

The algorithm constructs and solves a parity game that characterises satisfiability of the input formula χ. In this game one player attempts to construct a tableau structure for χ while the opposing player attempts to refute the existence of such a structure. Modal steps in this tableau construction are treated by using instances of the one-step satisfiability problem for the logic at hand, thereby generalising traditional modal tableau rules. The winning condition of the game is encoded by a non-deterministic parity automaton A_χ, reading infinite words that encode sequences of step-wise formula evaluations (so-called *formula traces*) within a coalgebra; such words encode branches in the constructed tableau structure. Conjunctions give rise to nondeterminism in this automaton, and the parity condition of the automaton is used to accept exactly those words that encode sequences of formula evaluations in which some least fixpoint is unfolded infinitely often. To use the language accepted by A_χ as the winning condition in a parity game, we transform A_χ to an equivalent deterministic parity automaton B_χ. This automaton then is paired with the tableau construction to yield a parity game in which the existential player aims to show the existence of a tableau structure in which all branches are rejected by B_χ, and that is built in such a way that modalities always are jointly one-step satisfiable. To ensure the latter property, the modal moves in the game invoke instances of the one-step satisfiability problem of the base logic. For more details on one-step satisfiability and the overall algorithm, see [17,21].

Corollary 2.2 ([21]). *Suppose that the one-step satisfiability problem is tractable. Then the satisfiability problem of the corresponding instance of the coalgebraic μ-calculus is in* ExpTime.

Remark 2.3. As mentioned above, previous algorithms for the coalgebraic μ-calculus (also implemented in COOL 2) rely on complete sets of modal tableau rules, specifically on one-step cutfree complete sets of so-called *one-step rules* [41]; such rules (in their incarnation as tableau rules) have a premiss with exactly one layer of modal operators and a purely propositional conclusion. A typical example is the usual tableau rule for the modal logic K: 'To satisfy $\Box a_1 \wedge \cdots \wedge \Box a_n \wedge \neg\Box a_0$, satisfy $a_1 \wedge \cdots \wedge a_n \wedge \neg a_0$'. It has been shown that the existence of a tractable one-step cutfree complete set of one-step rules implies tractability of one-step satisfiability [29], i.e. the approach via one-step satisfiability is more general.

As indicated in the introduction, a tractable one-step cutfree complete set of one-step rules for graded modal logic has been claimed in the literature [41,43] but has since turned out to be incomplete; we give a counterexample in the full version [17]. (A similar rule for Presburger modal logic [28] has also been shown to be in fact incomplete [29].)

3 Implementation

The previous version COOL [15] only implements fixpoint-free (coalgebraic) logics, such as standard modal logic, probabilistic modal logic, or coalition logic. The main novelty of the new version COOL 2, described here, is

- the addition of fixpoint constructs to the previously implemented logics, supporting alternation-free and aconjunctive fragments of the resulting μ-calculi, and implementing on-the-fly solving to allow early termination

– support for treating modal steps both by tableaux rules (when a suitable rule set exists), and by one-step satisfiability checking (in the remaining cases)

In more detail, COOL 2 is written in OCaml and implements the satisfiability checking algorithm described in Sect. 2, treating modal steps by solving instances of the one-step satisfiability problem[1]. For logics where a suitable set of modal tableau rules is implemented, those are used for the treatment of modal steps, rather than relying on one-step satisfiability (unless the user explicitly chooses otherwise); in these cases, COOL 2 essentially implements the algorithm described in [29]. The current implementation supports the alternation-free and the aconjunctive fragments of the standard μ-calculus (both serial and non-serial), the monotone μ-calculus [19], the alternating-time μ-calculus (i.e. coalition logic with fixpoint operators), and the graded μ-calculus. Tractable tableaux rules are available for all cases except for the graded μ-calculus, for which COOL 2 uses the one-step satisfiability algorithm to decide satisfiability. In particular, COOL 2 is the only existing reasoner for the graded μ-calculus (as well as the only reasoner covering the alternating-time μ-calculus beyond ATL).

The concrete logic used can be selected via a command-line parameter setting up the data structures in COOL 2 accordingly before parsing and checking the syntax of the given formula χ. COOL 2 then builds the determinised automaton B_χ, yielding the parity game described above in a step-wise manner, repeatedly adding nodes in *expansion steps* that explore the game. In the case of simpler alternation-free formulas, the Miyano-Hayashi method [31] is used to construct B_χ, resulting in asymptotically smaller games with a Büchi winning condition; for the more involved aconjunctive formulas, the implementation uses the permutation method for determinisation of limit-deterministic parity automata [9,22]. Nodes in the constructed game are marked as either unexpanded, undecided, unsatisfiable, or satisfiable.

Optional *solving steps* may take place at any point during the construction of B_χ, depending on runtime parameters of COOL 2; these steps compute the winning regions of the partial game that has been constructed so far and accordingly mark nodes as satisfiable or unsatisfiable, if possible. The reasoner terminates as soon as the initial node is marked satisfiable or unsatisfiable. If this does not allow for early termination, the game eventually becomes fully explored, at which point a final (obligatory) solving step for the complete game is guaranteed to mark the initial node, thereby ensuring termination.

We detail the implementation of the two main procedures within COOL 2.

Implementation of Expansion Steps. The propositional expansion steps in the game construction for nodes v are performed using the propositional satisfiability solver MiniSat [8] to compute a word that encodes consistent propositional formula manipulations for v. Afterwards, the successor of v in B_χ under this word is computed and added to the game.

When the one-step satisfiability based algorithm of COOL 2 is used, modal expansion steps for nodes v create new game nodes for each subset κ of the

[1] Sources are available at https://git8.cs.fau.de/software/cool.

modalities that are to be jointly satisfied at v; this is done by computing the successor of v in B_χ that is reached by manipulating each formula from κ.

When the tableau-based algorithm of COOL 2 is used, the modal expansion step for a node v instead computes all applications of a modal rule matching v and inserts, for each such rule application, and each conjunctive clause κ in the conclusion of the rule application, the new game node that is reached from v in B_χ by manipulating the modalities that constitute κ. Intuitively, using tableau rules reduces the search space by only adding nodes found in the conclusion of some matching rule application.

Any node that is added by some expansion step is initially marked as undecided. Crucially, all expansion steps perform on-the-fly determinisation, that is, given a game node v and a word that encodes a sequence of formula manipulations, the newly added game node is computed using only the information stored in v.

Implementation of Solving Steps. A single solving step computes the winning regions in the parity game that has been constructed up to this point, and marks nodes accordingly. The game solving is done using either the parity game solver PGSolver [10] or a native implementation provided by COOL 2 that solves the game by fixpoint iteration.

If the one-step satisfiability-based algorithm is used, an assigned modal node v is satisfiable if its modalities are jointly one-step satisfiable in those successors of v that are satisfiable themselves. An enumerative representation of the game thus contains existential moves to all subsets Π of subsets of modalities of v that are sufficiently large for one-step satisfaction of the modalities of v, followed by universal moves to nodes induced by any $\kappa \in \Pi$; the full game thus is of doubly-exponential size. This can be avoided by inlining the modal steps, thereby evading the intermediate nodes Π. The winning region can then be computed in single-exponential time by using COOL 2's native fixpoint iteration over a function that computes the two-tiered modal steps in one go.

Decision procedures for the one-step satisfiability problems in the relational and the graded case are implemented in COOL 2 along the lines of the algorithms described in [21, Example 6] (in the graded case, nondeterministic guessing is replaced with a recursive search procedure).

If the algorithm based on modal tableau rules is used, the treatment of modal steps follows the tableaux-based algorithm that is given in [3]. States v are satisfiable if for all rule applications that match v, the conclusion of the application contains a conjunctive clause κ such that the node induced by κ is satisfiable.

COOL 2 also allows the user to specify the desired frequency of optional game solving steps, including the options once and adaptive. With the option once, no intermediate solving takes place so that the game is fully constructed and solved just once, at the very end of the execution. With the option adaptive, intermediate solving takes places, but the frequency of solving reduces as the size of the constructed graph increases; this option implements *on-the-fly* solving and allows for finishing early in cases where a small model or refutation exists.

4 Evaluation

We conduct experiments in order to evaluate the performance of the various algorithms implemented in COOL in comparison with each other, as well as in comparison with other tools (where applicable).[2] Complete definitions of all formula series used in the evaluation as well as additional experimental results can be found in the full version [17].

Experiments: In a first experiment, we compare COOL 2 with the established reasoner FaCT++, which supports the description logic $\mathcal{SROIQ}(\mathcal{D})$ (subsuming fixpoint-free graded modal logic), using the following series of formulas from Snell et al. [43].

$$\mathsf{Cardinality}(n) := \langle n-1\rangle\neg p \wedge \langle n-1\rangle p \wedge [n]\neg q \wedge [n]q \qquad \text{(Sat)}$$
$$\mathsf{CardinalityU}(n) := \langle n-1\rangle\neg p \wedge \langle n-1\rangle p \wedge [n]\neg q \wedge [n-1]q \qquad \text{(UnSat)}$$

Intuitively, the satisfiable $\mathsf{Cardinality}(n)$ formulas express that there are at least $2n$ successors and that both q and $\neg q$ are satisfied in at most n successors, each; similarly the unsatisfiable $\mathsf{CardinalityU}(n)$ formulas state that there are at least $2n$ successors, and that q and $\neg q$ hold in at most n and $n-1$ successors, respectively; the latter statements imply that there are at most $2n-1$ successors, yielding a contradiction.

Going beyond next-step formulas, we continue by devising various complex series of graded μ-calculus formulas that involve (nested) fixpoints and express non-trivial properties of graded trees, automata and games.

- We obtain a series of unsatisfiable formulas by requiring the existence of an $n+1$-branching tree in which p holds everywhere while at the same time requiring that this tree contains some state with $n+2$ successors that satisfy p:

$$\mathsf{TreeU}(n) = (\nu X.\, \langle n\rangle(p \wedge X) \wedge [n+1]\neg p) \wedge (\mu Y.\, \langle n+1\rangle p \vee \langle n\rangle(p \wedge Y)) \qquad \text{(UnSat)}$$

- Next we turn our attention to graded formulas involving parity conditions. We devise a series of valid formulas expressing that graded parity automata can be transformed to graded Büchi automata accepting a superlanguage of the original automaton:

$$\mathsf{ParityToBuechi}(n,k) := \mathsf{Parity}(n,k) \rightarrow \mathsf{Buechi}(n,k) \qquad \text{(Valid)}$$

  Here, $\mathsf{Parity}(n,k)$ encodes parity acceptance with k priorities and grade n while $\mathsf{Buechi}(n,k)$ expresses Büchi acceptance by a nondeterministic automaton that eventually guesses the maximal priority that occurs infinitely often; the negated formula $\neg\mathsf{ParityToBuechi}(n,k)$ is unsatisfiable.

[2] Scripts and executables that allow for reproducing our experiments can be found at DOI 10.5281/zenodo.8042581.

- Rabin conditions are given by families of pairs $\langle i_j, f_j\rangle_{j\leq k}$ of sets i_j, f_j of states, and express the constraint that there is some $j \leq k$ such that states from i_j (*infinite*) are visited infinitely often and states from f_j (*finite*) are visited only finitely often. We can express Rabin conditions with k pairs (and one-step property ψ), Büchi properties and satisfaction of single Rabin-pairs by formulas $\mathsf{Rabin}(k, \psi)$, $\mathsf{Buechi}(f, \psi)$ and $\mathsf{RabinPair}(i, f, \psi)$, respectively. Then we obtain valid formulas stating that the existence of an $n+1$-branching tree that satisfies the Rabin condition on each path implies that there is a path satisfying a simpler Büchi condition or a single Rabin-pair, respectively:

$$\mathsf{RabinToBuechi}(k, n) := \mathsf{Rabin}(k, \langle n\rangle) \to \mathsf{Buechi}(i_1 \vee \ldots \vee i_k, \langle 0\rangle) \qquad \text{(Valid)}$$

$$\mathsf{RabinToRPair}(k, n) := \mathsf{Rabin}(k, \langle n\rangle) \to \bigvee_{1\leq j\leq k} \mathsf{RabinPair}(i_j, f_j, \langle 0\rangle) \qquad \text{(Valid)}$$

- Coming to games, we specify the winning regions in graded Büchi and Rabin games by formulas $\mathsf{BuechiG}(f, n)$ and $\mathsf{RabinG}(k, n)$, respectively; in such graded games, players are required to have at least n winning moves at their nodes in order to win. The following valid formulas then express that winning strategies in graded Rabin games with k pairs guarantee that some node from $i_1 \cup \ldots \cup i_k$ is visited infinitely often:

$$\mathsf{RabinGame}(k, n) := \mathsf{RabinG}(k, n) \to \mathsf{BuechiG}(i_1 \vee \ldots \vee i_k, n) \qquad \text{(Valid)}$$

In a final experiment on alternating-time formulas, we compare COOL 2 with TATL [6] on the ATL example formulas given in [6] as well as on additional formula series. For instance, we turn the formula $\langle\langle 1\rangle\rangle Gp \wedge \neg\langle\langle 2\rangle\rangle F \langle\langle 1\rangle\rangle Gp$ (written here using ATL syntax) from [6] into a series $\mathsf{Nest}(n)$ with increasing number of nested operators; formulas then alternatingly are satisfiable and unsatisfiable:

$$\chi(0) = p \qquad \chi(i+1) = \neg\langle\langle 2\rangle\rangle F \langle\langle 1\rangle\rangle G\chi(i) \qquad \mathsf{Nest}(n) = \langle\langle 1\rangle\rangle Gp \wedge \chi(n),$$

Results: We conducted all experiments on a virtual machine with four 2,3GHz vCPUs processors and 8GB of RAM. We compare with a 64-bit binary of FaCT++ v1.6.5 and with TATL. We compute all results with a timeout of 60 seconds and average the results over multiple executions. For the execution and measurement we use hyperfine[3]. Below, 'COOL' and 'COOL on-the-fly' refer to invoking COOL 2 with solving rate once and adaptive, respectively.

Results for the Cardinality and CardinalityU series are shown in Fig. 1 and Fig. 2, respectively. From $n = 10$ and $n = 8$ onwards, COOL 2 outperforms FaCT++ considerably. An explanation for this could be that FaCT++ appears to treat multiplicities in a naïve way while COOL 2 employs the more efficient one-step satisfiability algorithm.

Results for the unsatisfiable tree property are shown in Fig. 3. As these formulas contain fixpoint operators, a comparison with FaCT++ is not possible. While COOL 2 is generally capable of handling quite large branching factors, this experiment showcases the drawbacks of on-the-fly solving in the case that a formula cannot be decided early so that repeated attempts of solving the game early lead to overhead computations.

[3] https://github.com/sharkdp/hyperfine.

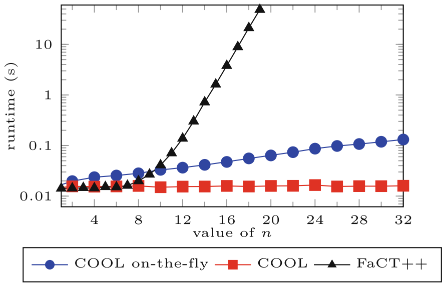


Fig. 1. Runtimes for Cardinality(n)

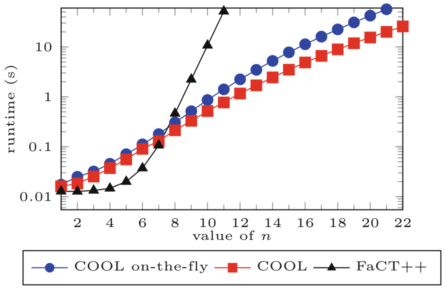


Fig. 2. Runtimes for CardinalityU(n)

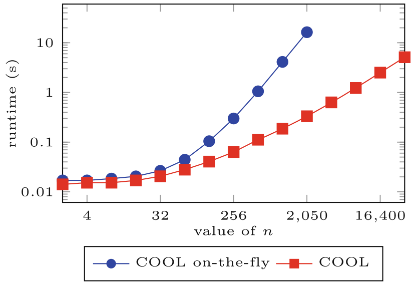


Fig. 3. Runtimes for TreeU(n)

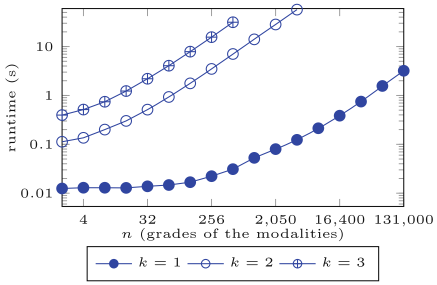


Fig. 4. Runtimes for ¬ParityToBuechi(n, k)

Runtimes for COOL 2 (using on-the-fly solving) on the unsatisfiable series of parity formulas ¬ParityToBuechi(n, k) are shown in Fig. 4. The results indicate that increasing the number of priorities k has a much stronger effect on the runtime than increasing multiplicities n in the modalities. This is in accordance with expectations as increasing k leads to much larger determinized automata and resulting satisfiabilty games, while increasing n only complicates the modal steps in the game while leaving the global game structure unchanged.

Results for the Rabin families of formulas are given in the table below, with † indicating a timeout of 60 s. COOL 2 is able to handle reasonably large formulas describing Rabin properties of automata and games, with the series for $n = 1$ expressing properties of standard automata (solved using tableau rules), and the series with $n = 2$ properties of graded automata with multiplicity 2 (solved using one-step satisfiability).

In accordance with previous experiments on random ATL formulas of larger sizes in [23], COOL 2 generally outperforms TATL by a large margin, starting from formulas containing at least five modalities or involving nesting of temporal operators; this trend is confirmed by Fig. 5 which shows the stepped execution times for the series Nest that alternates between being satisfiable and unsatisfiable

series\\k	1	2	3
RabinToBuechi$(k, 1)$	0.03	0.51	45.25
RabinToBuechi$(k, 2)$	0.08	10.56	†
RabinToRPair$(k, 1)$	0.03	8.38	†
RabinToRPair$(k, 2)$	0.07	†	†
RabinGame$(k, 1)$	0.05	1.04	†
RabinGame$(k, 2)$	0.31	43.94	†

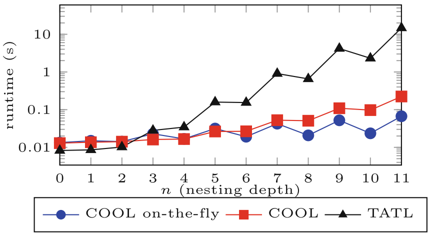


Fig. 5. Runtimes for the ATL series Nest(n)

In summary, COOL 2 shows promising performance in comparison to TATL and FaCT++, as well as for practical applicability. On graded formulas without fixpoints, COOL 2 scales much better than FaCT++ with regard to increasing multiplicities. In the presence of fixpoints, COOL 2 still scales well and can handle multiplicities that should be sufficient for practical use. The formula series ¬ParityToBuechi appears to show the limits of COOL 2 with the current implementation of graded one-step satisfiability checking. Nonetheless, our results indicate that COOL 2 is capable of automatically proving or refuting involved properties of (graded) ω-automata and games in reasonable time.

5 Conclusion

We have described and evaluated the current version COOL 2 of the *CO*algebraic *O*ntology *L*ogic reasoner (COOL). Future development will include the implementation of additional instance logics, such as the probabilistic and graded μ-calculus with linear inequalities, as well as support for the full coalgebraic μ-calculus via on-the-fly determinisation of *unrestricted* Büchi automata, using the Safra-Piterman construction.

References

1. Alur, R., Henzinger, T., Kupferman, O.: Alternating-time temporal logic. J. ACM **49**, 672–713 (2002). https://doi.org/10.1145/585265.585270
2. Calin, G., Myers, R., Pattinson, D., Schröder, L.: CoLoSS: the coalgebraic logic satisfiability solver. In: Methods for Modalities, M4M–5. ENTCS, vol. 231, pp. 41–54. Elsevier (2009). https://doi.org/10.1016/j.entcs.2009.02.028
3. Cîrstea, C., Kupke, C., Pattinson, D.: EXPTIME tableaux for the coalgebraic μ-calculus. Log. Meth. Comput. Sci. **7** (2011). https://doi.org/10.2168/LMCS-7(3:3)2011
4. Cîrstea, C., Kurz, A., Pattinson, D., Schröder, L., Venema, Y.: Modal logics are coalgebraic. Comput. J. **54**, 31–41 (2011). https://doi.org/10.1093/comjnl/bxp004
5. D'Agostino, G., Visser, A.: Finality regained: a coalgebraic study of Scott-sets and multisets. Arch. Math. Logic **41**, 267–298 (2002). https://doi.org/10.1007/s001530100110

6. David, A.: TATL: implementation of ATL tableau-based decision procedure. In: Galmiche, D., Larchey-Wendling, D. (eds.) TABLEAUX 2013. LNCS (LNAI), vol. 8123, pp. 97–103. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40537-2_10
7. David, A.: Deciding ATL* satisfiability by tableaux. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 214–228. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_14
8. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
9. Esparza, J., Kretínský, J., Raskin, J., Sickert, S.: From linear temporal logic and limit-deterministic Büchi automata to deterministic parity automata. Int. J. Softw. Tools Technol. Transf. **24**(4), 635–659 (2022). https://doi.org/10.1007/s10009-022-00663-1
10. Friedmann, O., Lange, M.: The PGSolver collection of parity game solvers. Technical report, LMU Munich (2009)
11. Friedmann, O., Lange, M.: A solver for modal fixpoint logics. In: Methods for Modalities, M4M–6 2009. ENTCS, vol. 262, pp. 99–111 (2010). https://doi.org/10.1016/j.entcs.2010.04.008
12. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: HermiT: an OWL 2 reasoner. J. Autom. Reason. **53**(3), 245–269 (2014). https://doi.org/10.1007/s10817-014-9305-1
13. Goré, R., Kupke, C., Pattinson, D., Schröder, L.: Global caching for coalgebraic description logics. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 46–60. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_5
14. Goré, R., Thomson, J., Widmann, F.: An experimental comparison of theorem provers for CTL. In: Temporal Representation and Reasoning, TIME 2011, pp. 49–56. IEEE (2011). https://doi.org/10.1109/TIME.2011.16
15. Gorín, D., Pattinson, D., Schröder, L., Widmann, F., Wißmann, T.: Cool – a generic reasoner for coalgebraic hybrid logics (system description). In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 396–402. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_31
16. Göttlinger, M., Schröder, L., Pattinson, D.: The alternating-time μ-calculus with disjunctive explicit strategies. In: Baier, C., Goubault-Larrecq, J. (eds.) Computer Science Logic, CSL 2021. LIPIcs, vol. 183, pp. 26:1–26:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.CSL.2021.26
17. Görlitz, O., Hausmann, D., Humml, M., Pattinson, D., Prucker, S., Schröder, L.: Cool 2 - a generic reasoner for modal fixpoint logics (2023). https://arxiv.org/abs/2305.11015
18. Haarslev, V., Möller, R.: RACER system description. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS, vol. 2083, pp. 701–705. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45744-5_59
19. Hansen, H.H., Kupke, C., Marti, J., Venema, Y.: Parity games and automata for game logic. In: Madeira, A., Benevides, M. (eds.) DALI 2017. LNCS, vol. 10669, pp. 115–132. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73579-5_8
20. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Asp. Comput. **6**, 512–535 (1994). https://doi.org/10.1007/BF01211866

21. Hausmann, D., Schröder, L.: Optimal satisfiability checking for arithmetic μ-calculi. In: Bojańczyk, M., Simpson, A. (eds.) FoSSaCS 2019. LNCS, vol. 11425, pp. 277–294. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17127-8_16
22. Hausmann, D., Schröder, L., Deifel, H.-P.: Permutation games for the weakly aconjunctive μ-calculus. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 361–378. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_21
23. Hausmann, D., Schröder, L., Egger, C.: Global caching for the alternation-free coalgebraic μ-calculus. In: Concurrency Theory, CONCUR 2016. LIPIcs, vol. 59, pp. 34:1–34:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.34
24. Kozen, D.: Results on the propositional μ-calculus. Theor. Comput. Sci. **27**, 333–354 (1983). https://doi.org/10.1016/0304-3975(82)90125-6
25. Kupferman, O., Piterman, N., Vardi, M.Y.: Fair equivalence relations. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 702–732. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39910-0_30
26. Kupferman, O., Sattler, U., Vardi, M.Y.: The complexity of the graded μ-calculus. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 423–437. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45620-1_34
27. Kupke, C., Marti, J., Venema, Y.: Size measures and alphabetic equivalence in the μ-calculus. In: Baier, C., Fisman, D. (eds.) Logic in Computer Science, LICS 2022, pp. 18:1–18:13. ACM (2022). https://doi.org/10.1145/3531130.3533339
28. Kupke, C., Pattinson, D.: On modal logics of linear inequalities. In: Advances in Modal Logic, AiML 2010, pp. 235–255. College Publications (2010)
29. Kupke, C., Pattinson, D., Schröder, L.: Coalgebraic reasoning with global assumptions in arithmetic modal logics. ACM Trans. Comput. Log. **23**(2), 11:1–11:34 (2022). https://doi.org/10.1145/3501300
30. Larsen, K., Skou, A.: Bisimulation through probabilistic testing. Inform. Comput. **94**, 1–28 (1991). https://doi.org/10.1016/0890-5401(91)90030-6
31. Miyano, S., Hayashi, T.: Alternating finite automata on ω-words. Theor. Comput. Sci. **32**, 321–330 (1984). https://doi.org/10.1016/0304-3975(84)90049-5
32. Nalon, C., Zhang, L., Dixon, C., Hustadt, U.: A resolution prover for coalition logic. In: Mogavero, F., Murano, A., Vardi, M.Y. (eds.) Strategic Reasoning, SR 2014. EPTCS, vol. 146, pp. 65–73 (2014). https://doi.org/10.4204/EPTCS.146.9
33. Niwiński, D.: On fixed-point clones. In: Kott, L. (ed.) ICALP 1986. LNCS, vol. 226, pp. 464–473. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-16761-7_96
34. Parikh, R.: Propositional game logic. In: Foundations of Computer Science, FOCS 1983. IEEE Computer Society (1983). https://doi.org/10.1109/SFCS.1983.47
35. Pattinson, D.: Expressive logics for coalgebras via terminal sequence induction. Notre Dame J. Formal Logic **45**, 19–33 (2004). https://doi.org/10.1305/ndjfl/1094155277
36. Pauly, M.: A modal logic for coalitional power in games. J. Logic Comput. **12**, 149–166 (2002). https://doi.org/10.1093/logcom/12.1.149
37. Peleg, D.: Concurrent dynamic logic. J. ACM **34**, 450–479 (1987). https://doi.org/10.1145/23005.23008
38. Rutten, J.: Universal coalgebra: a theory of systems. Theor. Comput. Sci. **249**, 3–80 (2000). https://doi.org/10.1016/S0304-3975(00)00056-6
39. Schewe, S.: Synthesis of distributed systems. Ph.D. thesis, Universität des Saarlands (2008)

40. Schröder, L.: Expressivity of coalgebraic modal logic: the limits and beyond. Theor. Comput. Sci. **390**(2–3), 230–247 (2008). https://doi.org/10.1016/j.tcs.2007.09.023
41. Schröder, L., Pattinson, D.: PSPACE bounds for rank-1 modal logics. ACM Trans. Comput. Log. **10**(2), 13:1–13:33 (2009). https://doi.org/10.1145/1462179.1462185
42. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: a practical OWL-DL reasoner. J. Web Semant. **5**(2), 51–53 (2007). https://doi.org/10.1016/j.websem.2007.03.004
43. Snell, W., Pattinson, D., Widmann, F.: Solving graded/probabilistic modal logic via linear inequalities (system description). In: Bjørner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol. 7180, pp. 383–390. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_30
44. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: system description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 292–297. Springer, Heidelberg (2006). https://doi.org/10.1007/11814771_26

Choose Your Colour: Tree Interpolation for Quantified Formulas in SMT

Elisabeth Henkel[1](✉), Jochen Hoenicke[2], and Tanja Schindler[3]

[1] University of Freiburg, Freiburg im Breisgau, Germany
henkele@informatik.uni-freiburg.de
[2] Certora, Tel Aviv-Yafo, Israel
jochen@certora.com
[3] University of Liège, Liège, Belgium
tanja.schindler@uliege.be

Abstract. We present a generic tree-interpolation algorithm in the SMT context with quantifiers. The algorithm takes a proof of unsatisfiability using resolution and quantifier instantiation and computes interpolants (which may contain quantifiers). Arbitrary SMT theories are supported, as long as each theory itself supports tree interpolation for its lemmas. In particular, we show this for the theory combination of equality with uninterpreted functions and linear arithmetic. The interpolants can be tweaked by virtually assigning each literal in the proof to interpolation partitions (colouring the literals) in arbitrary ways. The algorithm is implemented in SMTInterpol.

Keywords: Tree Interpolation · Quantified Formulas · SMT

1 Introduction

Craig interpolants [7] were originally proposed to reason about proof complexity. In the last two decades, research reignited when interpolants proved useful for software verification, in particular for generating invariants [15]. Tree interpolants are useful for verifying programs with recursion [12], and for solving non-linear Horn-clause constraints [23], which can be used for thread modular reasoning [10,16] and verifying array programs [20]. For many verification problems, reasoning about first-order quantified formulas is needed. Quantified formulas are, among others, needed to model unsupported theories or to express global properties of arrays [19], for example, sortedness [3,24].

An interpolation problem is an unsatisfiable conjunction of several input formulas, the partitions of the interpolation problem. An interpolant summarises the contribution of a single or multiple partitions to the unsatisfiability. Interpolants can be computed from resolution proofs. However, most methods require localised proofs where each literal is associated with some input partition [22]. Proofs generated by SMT solvers, especially with quantifier instantiations, usually contain mixed terms and literals created during the solving process that cannot be associated with a single input formula.

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 248–265, 2023.
https://doi.org/10.1007/978-3-031-38499-8_15

In this paper, we extend our work on proof tree preserving sequence interpolation of quantified formulas [13]. The method presented therein allows for the computation of inductive sequence interpolants from instantiation-based resolution proofs of quantified formulas in the theory of uninterpreted functions. The key idea of this method is to perform a virtual modification of mixed terms introduced through quantifier instantiations, thus allowing to compute an inductive sequence of interpolants on a single, non-local proof tree.

We extend the interpolation algorithm to compute tree interpolants and to support arbitrary SMT theories (with the single restriction that such a theory itself must support tree interpolation for its lemmas). We simplify the treatment of mixed terms by virtually flattening all literals independently of the partitioning. We show that the literals can be coloured (assigned to a partition) arbitrarily, and that for every colouring, correct interpolants are produced. The interpolants contain quantifiers for the flattening variables that bridge different partitions, and by choosing colours sensibly the number of quantifiers can be reduced. In contrast to previous works [1,12] which produce tree interpolants by repeated binary interpolation and require multiple proofs, our method computes a tree interpolant from a single proof.

Related Work. Many practical algorithms to compute interpolants have been presented. We focus here on proof-based methods that either work in the presence of quantifiers, or that can compute tree interpolants, or both.

Our work builds on the method presented in [4] for computing interpolants from instantiation-based proofs in SMT. It is based on *purifying* quantifier instantiations by introducing variables for terms not fitting the partition, and adding defining auxiliary equalities as a new input clause in the proof. Our method introduces these variables and equalities only virtually for computing the partial interpolants. Thus, tree interpolants can be computed from a single proof of unsatisfiability, while in [4] a purified proof is required for each partition.

There exist several methods to compute interpolants for quantified formulas inductively from superposition-based proofs. In [2], each literal is given a *label* (similar to our colouring) used to project the clause to the different partitions. First, a *provisional* interpolant is computed that may contain local symbols. These symbols are replaced by quantified variables to obtain an interpolant. In contrast to our method, the approach only works when the provisional interpolants contain at most local constants, i.e., no local functions or predicates, and the assignment of labels is not flexible as our colouring. The method in [17] is based on a slightly modified proof, where substitution steps are done separately. First, a *relational* interpolant is computed, which may contain local function symbols, but only shared predicates. In logic without equality, or when the only local symbols are constants, the relational interpolant can be turned into an interpolant by quantifying over non-shared terms, respecting their dependencies.

A very different method based on summarising subproofs is presented in [9]. The proof is split into subproofs belonging to a single partition. The relevant subproofs are summarised in an *intermediant* stating that their premises imply their conclusion. If the subproofs contain only symbols of the respective partition, the resulting formula is an interpolant. If the proof can be split in that way, the method works for any theory and proof system, but for tree interpolation, a different proof would be required for each partitioning.

Tree interpolants can be computed by repeated binary interpolation from formulas where the children interpolants are included, as discussed in [12]. In the propositional setting, [11] discusses under which conditions sets of interpolants with certain relations, such as tree interpolants, can be obtained by binary interpolation on different partitionings of the same formula. The method is implemented in OpenSMT, but the solver, and therefore the interpolation engine, does not support quantifiers.

A general framework for computing tree interpolants for ground formulas from a single proof has been presented in [5]. It works for combinations of equality-interpolating theories and is based on projecting *mixed literals* using auxiliary variables and predicates. Additionally, the rule for computing a resolvent's interpolant from its antecedents' interpolants is more involved. The method cannot deal with quantifier instantiations, nor with terms mixing subterms from different partitions. We discuss in Sect. 6 how it can be combined with the interpolation method for quantified formulas presented in this paper.

The first implementation of a tree interpolation algorithm in the presence of quantifiers and theories was in Vampire [1]. It is based on repeatedly computing binary interpolants for modified interpolation problems, similar to [12]. For each binary computation, the proof must be localised in order to be able to compute interpolants. In contrast, our method computes tree interpolants in one go from a single proof that has been obtained without knowledge of the partitioning of the tree interpolation problem. To the best of our knowledge, Vampire is the only other tool that is able to compute tree interpolants in the presence of quantifiers.

2 Notation

We assume that the reader is familiar with first-order logic. We define a *theory* $\mathcal{T}$ by its *signature*, that contains constant, function and predicate symbols, and its set of *axioms*, closed formulas that fix the meaning of those function and predicate symbols that are *interpreted* by the theory.

A *term* is a variable or the application of an n-ary function symbol to n terms. An *atom* is the application of an n-ary predicate to n terms, and a *literal* is an atom or its negation. A *clause* is a disjunction of literals, and a formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. We use $\top$ (resp. $\bot$) for the formula that is always true (resp. false).

We will demonstrate our algorithm using the theory of equality, and the theory of linear arithmetic (with rationals and/or integers). The theory of equality establishes reflexivity, symmetry, and transitivity of the equality predicate $=$, and congruence for each *uninterpreted* function symbol. For simplicity of the presentation, uninterpreted constants are considered as 0-ary functions, and uninterpreted predicate symbols as uninterpreted functions with Boolean return value. The theory of linear arithmetic contains the predicates $\leq, <$, rational constants c, the binary addition function $+$, and a family of unary multiplication functions $c\cdot$, one for each rational constant c. These symbols have their usual semantics, and the main theory lemmas are trichotomy $(x < y \lor x = y \lor x > y)$ and a variant of Farkas lemma. For simplicity, we apply arithmetic conversions implicitly and treat $x \leq y$ and $y \geq x$ and $1 \cdot x + (-1) \cdot y \leq 0$ as the same literal, and $x > y$ as its negated literal.

We denote constants by a, b, c, functions by f, g, h, variables by v, x, y, z, and terms by s, t. We use ℓ for literals, C for clauses, and ϕ, F, I for formulas.

For a term t, the outermost (or *head*) function symbol is denoted by $hd(t)$. The set of all uninterpreted function symbols occurring in a formula F is $symb(F)$ and the set of all free variables in F is $FreeVars(F)$. The result of substituting in a formula F each occurrence of a variable x by a term t is denoted by $F\{x \mapsto t\}$. By $\bar{x}$ and $\bar{t}$, we denote the list of variables $x_1, \ldots, x_n$ and terms $t_1, \ldots t_n$, respectively. We use the symbol $\equiv$ to denote equivalence between formulas, and to assign a formula to a formula variable.

3 Preliminaries

Craig Interpolation. A binary *Craig interpolant* [7] for an unsatisfiable conjunction $A \land B$ is a formula I that is implied by A, contradicts B, and contains only symbols that occur in both A and B. A generalisation are tree interpolants, which introduce several partitions in a tree-like structure.

Definition 1 (Tree interpolation). *A* tree interpolation problem (V, E, F) *is a labelled binary tree where V is a set of nodes connected by directed edges $E \subseteq V \times V$ pointing towards the root node. Every node except for the root node has one outgoing edge to its parent, and each non-leaf node has exactly two incoming edges. The* partitions $P \subseteq V$ *of the tree interpolation problem are the leaf nodes. The labelling function F assigns a formula to each partition $p \in P$ of the tree such that their conjunction is unsatisfiable. We use $st(v) \subseteq P$ to denote the set of leaves in the subtree of the node v, i.e., the set of leaves for which a path to the node v exists.*

A tree interpolant *for the interpolation problem (V, E, F) is a labelling function I for all nodes with the following properties:*

1. *The label $I(v_r)$ of the root node v_r of the tree is $\perp$.*
2. *For each leaf node $p \in P$, its interpolant $I(p)$ is implied by the formula $F(p)$.*
3. *For each inner node $v \in V \setminus P$, its interpolant $I(v)$ is implied by the conjunction $I(v_1) \land I(v_2)$ of the interpolants labelled to the two child nodes v_1, v_2.*
4. *For each node v, the symbols in $I(v)$ occur both inside and outside the subtree $st(v)$, i.e., $symb(I(v)) \subseteq (\bigcup_{p \in st(v)} symb(F(p))) \cap (\bigcup_{p \notin st(v)} symb(F(p)))$.*

Remarks. In contrast to the earlier definition of tree interpolation [1,5], only the leaves of the tree are labelled by F here. A tree interpolation problem with labelled inner nodes can be transformed to our formalism by adding a leaf child to each such node. A non-binary tree can be extended to a binary tree by adding more internal nodes. If the interpolants of the newly created nodes are ignored, the remaining interpolants are tree interpolants according to the earlier definition for tree interpolation.

A binary interpolant of A and B corresponds to the tree interpolant of the tree containing just two leaves A and B, more precisely, it is the interpolant labelled to the first leaf. Vice versa, each interpolant $I(v)$ of a tree interpolant is also a binary interpolant of the formulas in the partitions $A := st(v)$ and $A^c := P \setminus st(v)$. Since the set A defines v uniquely, we can also use I_A to denote $I(v)$. We call a symbol *A-local* if it only occurs in partitions in A, *A^c-local* if it only occurs in partitions in A^c, and *shared* if it occurs in both. The interpolant may only contain shared symbols.

Theory Combination. We assume that the solver uses Nelson–Oppen style theory combination sharing equalities without explicitly introducing auxiliary variables, and that each lemma in the proof belongs to one theory. Subterms in these lemmas containing symbols from a different theory are treated as if they were auxiliary variables. We further assume that there is a theory-specific interpolation procedure for the lemmas. In this paper, we do not have the assumption that theories are equality-interpolating. We introduce quantifiers in the interpolants for such theories. However, our approach can also be combined with equality-interpolating theories and corresponding procedures to avoid quantifiers, see Sect. 6.

CNF Transformation and Quantifiers. We assume that complex input formulas are transformed to CNF by Tseitin-encoding, which introduces Boolean proxy atoms. Existentially quantified variables are replaced with Skolem constants or functions (if nested under a universal quantifier) and conjunctions are lifted over universal quantifiers. Complex subformulas under a universal quantifier are replaced by uninterpreted predicates, taking as arguments the quantified variables. Quantified Tseitin-style axioms give the meaning for these predicates. Thus, we end up with quantified clauses of the form $\forall \bar{x}.\, \ell_1(\bar{x}) \vee \cdots \vee \ell_n(\bar{x})$, which we treat as a proxy literal. Instances of quantified clauses are created using instantiation lemmas of the form $\neg(\forall \bar{x}.\, \ell_1(\bar{x}) \vee \cdots \vee \ell_n(\bar{x})), \ell_1(\bar{t}), \ldots, \ell_n(\bar{t})$ where $\bar{t}$ are ground terms. Note that the proxy atom for a quantified formula occurs only positively in input clauses and negated in instantiation lemmas. We note that all preprocessing steps are done locally for each input formula, and that auxiliary predicates and Skolem functions are fresh predicate or function symbols. An interpolant of the preprocessed formulas is also an interpolant of the original formulas, because the auxiliary symbols are not shared between different input formulas and will never appear in the interpolant.

Proofs. A *resolution proof* for the unsatisfiability of a formula in CNF is a derivation of the empty clause $\bot$ using the resolution rule

$$\frac{C_1 \vee \ell \qquad C_2 \vee \neg \ell}{C_1 \vee C_2}$$

where C_1 and C_2 are clauses, and ℓ is a literal called the *pivot* (literal). A resolution proof can be represented by a tree, or more generally, if the same subproof is used more than once, by a directed acyclic graph (DAG). In our setting, the DAG has three types of leaves: *input clauses*, *theory lemmas*, i.e., clauses that are valid in the theory $\mathcal{T}$, and *instantiation lemmas* of the form $\neg(\forall \bar{x}.\phi(\bar{x})) \vee \phi(\bar{t})$. The inner nodes are clauses obtained by resolution, and the unique root node is the empty clause $\bot$.

Binary interpolants can be computed from a resolution proof by computing so-called partial interpolants for each clause. Each proof step proves a clause C as a consequence of the input $A \wedge B$, hence it proves that $A \wedge B \wedge \neg C$ is unsatisfiable. If each literal in the proof is assigned to, or *coloured* with, either partition A or B, a *partial interpolant* for each intermediate step is the interpolant of $A \wedge \neg C \downharpoonleft A$ and $B \wedge \neg C \downharpoonleft B$, where the projection $\neg C \downharpoonleft A$ extracts from the conjunction $\neg C$ all literals that are coloured with partition A. McMillan showed for propositional logic that partial interpolants (cf. Definition 2 in [18]) can be computed recursively for each resolution step as the disjunction of the partial interpolants of the antecedents if the pivot is coloured as A, and their conjunction if it is coloured as B.

4 Colouring of Terms and Literals

In this section, we fix an interpolation problem (V, E, F), with partitions $P \subseteq V$. We use the following example to illustrate our interpolation algorithm.

Example 1 (Running example). Take the tree interpolation problem with nodes $V = \{123, 1, 23, 2, 3\}$ and edges $E = \{(1, 123), (23, 123), (2, 23), (3, 23)\}$ (see also Fig. 1), where the partitions $P = \{1, 2, 3\}$ are labelled with $F(p) \equiv \phi_p$ where

$$\phi_1 \equiv \forall x.\ g(h(x)) \leq x, \quad \phi_2 \equiv \forall y.\ g(y) \geq b, \quad \phi_3 \equiv \forall z.\ f(g(z)) \neq f(b).$$

The conjunction of the three formulas is unsatisfiable. Instantiating ϕ_1 with b gives $g(h(b)) \leq b$. Instantiating ϕ_2 with $h(b)$ gives $g(h(b)) \geq b$. Together they imply $g(h(b)) = b$. However, this contradicts ϕ_3 instantiated with $h(b)$. This proof creates, among others, the new literal $g(h(b)) \leq b$. The term $g(h(b))$ contains function symbols that do not occur in a common partition.

We recall that by $symb(F(p))$, we denote the uninterpreted function symbols occurring in the formula $F(p)$. We also keep track of the partitions where a symbol occurs:

Definition 2 (Partitions). *The* partitions of a function symbol f *are the partitions where this symbol occurs:*

$$partitions(f) = \{p \in P \mid f \in symb(F(p))\}.$$

McMillan's interpolation algorithm assumes that all symbols of a literal occur in one partition, such that the literal can be coloured with that partition. This is no longer the case in SMT, because new literals are created during the proof search, especially in the presence of instantiation lemmas. Our solution to this problem is to split each literal into many smaller literals and assign each of them to a partition. To keep the presentation simple, we flatten all (non-proxy) literals using a fresh variable for each application term. Thus, for every term t occurring in the resolution proof, we create a fresh variable v_t and associate with it a set of flattening equalities. In each literal, the top-level terms are replaced with their associated variable, and the defining equalities are conjoined.

Definition 3 (Flattening). *For a term t, we introduce a fresh variable v_t, and similarly for all its subterms. The associated set of flattening equalities $\mathit{FlatEQ}(t)$ is defined as follows:*

$$\mathit{FlatEQ}(t) = \{v_{f(t_1,\dots,t_n)} = f(v_{t_1},\dots,v_{t_n}) \mid f(t_1,\dots,t_n) \textit{ is a subterm of } t\}.$$

The flattened version of a literal ℓ is

$$\mathit{flatten}(\ell) \equiv \begin{cases} v_{t_1} = v_{t_2} & \textit{if } \ell \equiv t_1 = t_2 \\ c_1 \cdot v_{t_1} + \cdots + c_n \cdot v_{t_n} \leq c & \textit{if } \ell \equiv c_1 \cdot t_1 + \cdots + c_n \cdot t_n \leq c \end{cases}$$

and the associated set of flattening equalities is as follows

$$\mathit{FlatEQ}(\ell) = \begin{cases} \mathit{FlatEQ}(t_1) \cup \mathit{FlatEQ}(t_2) & \textit{if } \ell \equiv t_1 = t_2 \\ \mathit{FlatEQ}(t_1) \cup \cdots \cup \mathit{FlatEQ}(t_n) & \textit{if } \ell \equiv c_1 \cdot t_1 + \cdots + c_n \cdot t_n \leq c. \end{cases}$$

The flattened version of a negated literal is the negation of the flattened literal, i.e., $\mathit{flatten}(\neg\ell) \equiv \neg\mathit{flatten}(\ell)$. The set of flattening equalities for a negated literal is the set of flattening equalities for the literal itself, i.e., $\mathit{FlatEQ}(\neg\ell) = \mathit{FlatEQ}(\ell)$.

The conjunction of the equalities in $\mathit{FlatEQ}(t)$ implies that $v_t = t$. Similarly, the conjunction $\mathit{flatten}(\ell) \wedge \bigwedge \mathit{FlatEQ}(\ell)$ implies the literal ℓ and is equisatisfiable to ℓ. Proxy literals like quantified formulas are not flattened, as they will never occur in a partial interpolant. For such a proxy literal, $\mathit{flatten}(\forall x.\phi(x)) \equiv \forall x.\phi(x)$ and $\mathit{FlatEQ}(\forall x.\phi(x)) = \emptyset$.

Example 2 (Flattening). Consider the literal $g(h(b)) \leq b$. Its flattened version is $\mathit{flatten}(g(h(b)) \leq b) \equiv v_{g(h(b))} \leq v_b$, and the set of flattening equalities is

$$\begin{aligned} \mathit{FlatEQ}(g(h(b)) \leq b) &= \mathit{FlatEQ}(g(h(b))) \cup \mathit{FlatEQ}(b) \\ &= \{v_{g(h(b))} = g(v_{h(b)}), v_{h(b)} = h(v_b), v_b = b\}. \end{aligned}$$

To define partial interpolants, we colour each atom ℓ with some partition, denoted by $\mathit{colour}(\ell) \in P$. The negated atom always has the same colour. For proxy atoms created during the CNF conversion, it is important to colour them with the input partition from which they were created. The colour of other literals can be chosen arbitrarily, but a good heuristic would choose a partition where most of the outermost function symbols occur. Each flattening equality is associated with all partitions where the corresponding function symbol occurs. The *projection* of auxiliary equations on a partition p, denoted by $\mathit{FlatEQ}(\ell) \downharpoonleft p$, is defined as the conjunction of the equalities $(v_{f(t_1,\ldots,t_n)} = f(v_{t_1},\ldots,v_{t_n})) \in \mathit{FlatEQ}(\ell)$ where $p \in \mathit{partitions}(f)$.

Finally, we define the projection of a literal ℓ to a partition p. The *projection kernel* $\ell \downharpoonleft^- p$ is $\mathit{flatten}(\ell)$ if $p = \mathit{colour}(\ell)$ or $\top$ otherwise. The *projection of* ℓ *to* p is defined as $\ell \downharpoonleft p \equiv \ell \downharpoonleft^- p \wedge \mathit{FlatEQ}(\ell) \downharpoonleft p$. We define the projection to a set of partitions $\ell \downharpoonleft A$ with $A \subseteq P$ (and similarly $\ell \downharpoonleft^- A$) as the conjunction of all projections $\ell \downharpoonleft p$ with $p \in A$. For a conjunction of literals $F \equiv \ell_1 \wedge \cdots \wedge \ell_n$, we define $F \downharpoonleft p \equiv \ell_1 \downharpoonleft p \wedge \cdots \wedge \ell_n \downharpoonleft p$ and similar for $F \downharpoonleft A$, $F \downharpoonleft^- p$ and $F \downharpoonleft^- A$.

Example 3 (Projection of literals). Consider again the literal $g(h(b)) \leq b$ from our running example (Example 1), and assume that we arbitrarily assign it to partition 2, i.e., $\mathit{colour}(g(h(b)) \leq b) = 2$. We have $\mathit{partitions}(g) = \{1, 2, 3\}$, $\mathit{partitions}(h) = \{1\}$ and $\mathit{partitions}(b) = \{2, 3\}$. The projections are hence:

$$\begin{aligned} g(h(b)) \leq b \downharpoonleft 1 &\equiv v_{g(h(b))} = g(v_{h(b)}) \wedge v_{h(b)} = h(v_b) \\ g(h(b)) \leq b \downharpoonleft 2 &\equiv v_{g(h(b))} \leq v_b \wedge v_{g(h(b))} = g(v_{h(b)}) \wedge v_b = b \\ g(h(b)) \leq b \downharpoonleft 3 &\equiv v_{g(h(b))} = g(v_{h(b)}) \wedge v_b = b \end{aligned}$$

Similar to the last paragraph in Sect. 3, we define a partial interpolant of a clause C as an interpolant of the input problem and $\neg C$. More precisely, it is the tree interpolant of a slightly modified tree interpolation problem, where the projection $\neg C \downharpoonleft p$ is added to each leaf node $p \in P$. Since this step adds flattening variables potentially shared between several partitions, these variables can occur in the interpolants. The following definition accounts for the variables occurring in the projection of a clause.

Definition 4 (Supported variable). *We call a variable* v_t supported by a clause C *if its corresponding term* t *is a subterm of a non-proxy literal* ℓ *in* C*.*

The partial tree interpolant of a clause C may then contain a variable v_t as long as it is supported by the clause C.

Definition 5 (Partial tree interpolant). *A* partial tree interpolant *for a clause* C *is a tree interpolant as defined in Definition 1 for the tree interpolation problem* (V, E, F') *where the leaves are labelled with* $F'(p) \equiv F(p) \wedge \neg C \downharpoonleft p$*. For the symbol condition, all variables supported by the clause may occur in all partial interpolants.*

5 Interpolation for Quantified Formulas

In the following, we describe how to compute tree interpolants for instantiation-based resolution proofs. We assume that each literal has been assigned to exactly one partition of the tree interpolation problem, as described in the previous section. Following McMillan's algorithm, we compute partial tree interpolants inductively over the proof tree. The leaves of the proof tree are theory lemmas, for which we use theory-specific interpolation procedures, or they are input clauses or instantiation lemmas, for which we compute partial tree interpolants as described below. The inner nodes are obtained by resolution steps, for which we follow McMillan's algorithm to combine interpolants, and additionally treat variables that violate the symbol condition, as described later in this section.

5.1 Interpolation Algorithm

We start by explaining how the interpolants for leaf nodes are computed. Our algorithm computes interpolants separately for each node $v \in V$ in the tree interpolation problem. As mentioned in the preliminaries, we set $A = st(v)$ and use I_A to denote the interpolant $I(v)$.

Input Clauses. We assume that each input clause occurs in exactly one partition. The partial tree interpolant for an input clause C from partition p is given by $I_A \equiv \neg(\neg C \downharpoonleft^{-} A^c)$ if $p \in A$, and $I_A \equiv \neg C \downharpoonleft^{-} A$ if $p \notin A$.

Note that the literals can be assigned to a different partition than the clause. Although it makes sense to assign a literal to the same partition as the input clause it occurs in, this is not possible when the literal occurs in several input clauses. Therefore, the above formulas are not necessarily $\top$ or $\bot$. Proxy literals always have the same colour as the input clause and will therefore never appear in the interpolant.

Instantiation Lemmas. The partial tree interpolant for an instantiation lemma C obtained from a quantified input clause $\forall x.\phi(x)$ from partition $colour(\forall x.\phi(x))$ is computed in the same way as for input clauses.

Theory Lemmas. We only require that for each theory one can compute a partial tree interpolant for its lemmas, or to be more precise, the flattened negated lemmas. Thus, we can reuse any existing procedure. For self-containment, we cover transitivity, congruence, trichotomy and Farkas lemmas, which are the kind of lemmas our solver produces for the theory of equality and linear arithmetic.[1]

For a *transitivity* lemma with the corresponding conflict $\neg C \equiv t_1 = t_2 \wedge \cdots \wedge t_{n-1} = t_n \wedge t_1 \neq t_n$ we can ignore the auxiliary equations introduced by flattening the terms, as the projection kernel is also a transitivity lemma. A partial tree interpolant is computed by summarising for each A the chains of the flattened equalities (and, if applicable, the single disequality) that are assigned to a partition $p \in A$. More precisely, let $i_1 < \cdots < i_m$ be the boundary indices such that $colour(t_{i_j-1} = t_{i_j}) \in A$ and $colour(t_{i_j} = t_{i_j+1}) \notin A$ or vice versa. Set $i_1 = 1$ if $t_1 \neq t_n$ and $t_1 = t_2$ are in different partitions and $i_m = n$ if $t_{n-1} = t_n$

[1] Branches in linear integer arithmetic [8] are decisions on inequality literals and are handled by our resolution rule.

and $t_1 \neq t_n$ are in different partitions. If $m = 0$, then all colours of the equalities are in A and the interpolant is $\bot$, or they are all in A^c and the interpolant is $\top$. Otherwise, the interpolant summarises the equalities between the boundary indices that have a colour in A: if $colour(t_1 = t_n) \notin A$, then the interpolant is $I_A \equiv v_{i_1} = v_{i_2} \wedge v_{i_3} = v_{i_4} \wedge \cdots \wedge v_{i_{m-1}} = v_{i_m}$, otherwise the interpolant is $I_A \equiv v_{i_2} = v_{i_3} \wedge \cdots \wedge v_{i_{m-2}} = v_{i_{m-1}} \wedge v_{i_m} \neq v_{i_1}$. Here, v_i denotes the auxiliary variable introduced for t_i.

The flattened version of the conflict corresponding to a *congruence* lemma $C \equiv f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n) \vee t_1 \neq s_1 \vee \cdots \vee t_n \neq s_n$ is

$$\begin{aligned}&v_{f(t_1,\ldots,t_n)} \neq v_{f(s_1,\ldots,s_n)} \wedge v_{t_1} = v_{s_1} \wedge \ldots \wedge v_{t_n} = v_{s_n}\\ &\wedge v_{f(t_1,\ldots,f_n)} = f(v_{t_1}, \ldots, v_{t_n}) \wedge v_{f(s_1,\ldots,s_n)} = f(v_{s_1}, \ldots, v_{s_n})\\ &\wedge \bigwedge\{\ell \mid \ell \in \mathit{FlatEQ}(t), t \in \{t_1, \ldots, t_n, s_1, \ldots, s_n\}\}.\end{aligned}$$

Note that the formula is still a congruence conflict if we drop the last line. Consequently, the flattening equalities for the arguments of the f-applications, and for their subterms, are not needed in the computation of a partial interpolant, they only establish the implication between the flattened and the original lemma. To obtain a partial tree interpolant, we first choose an arbitrary partition $p_f \in \mathit{partitions}(f)$. The partial tree interpolant is computed as follows.

$$I_A \equiv \begin{cases}\neg(\neg C \downharpoonleft^- A^c) & \text{if } p_f \in A\\ \neg C \downharpoonleft^- A & \text{otherwise}\end{cases}$$

For a *trichotomy* lemma $C \equiv t_1 = t_2 \vee t_1 > t_2 \vee t_1 < t_2$, both $I_A \equiv \neg C \downharpoonleft^- A$ and $I'_A \equiv \neg(\neg C \downharpoonleft^- A^c)$ are partial interpolants. We can always choose the projection that contains at most one literal.

A *Farkas* lemma has the form $C \equiv \neg(s_1 \leq b_1) \vee \cdots \vee \neg(s_n \leq b_n)$ where s_i is of the form $c_{i1} \cdot v_1 + \ldots + c_{im} \cdot v_m$ and b_i, c_{ij} are numeric (integer) constants. It is a valid lemma if there are Farkas coefficients (numeric integer constants) $k_1, \ldots, k_n > 0$ with $\sum_{i=1}^n k_i \cdot s_i = 0$ and $\sum_{i=1}^n k_i \cdot b_i < 0$. We assume that the lemma is flattened and all v_i are variables. The flattening equalities can be omitted from the lemma without changing its validity. For a set of partitions A, we denote by $L_A := \{i \mid colour(s_i \leq b_i) \in A\}$ the indices where $s_i \leq b_i$ is A-local. The partial tree interpolant for a Farkas lemma is computed by summing up the A-local literals multiplied by their Farkas coefficients. We obtain $I_A \equiv (\sum_{i \in L_A} k_i \cdot s_i) \leq (\sum_{i \in L_A} k_i \cdot b_i)$. Variables whose coefficients sum to zero are removed from the inequality. If A contains all inequalities, they sum up to the conflict $0 \leq \sum_{i=1}^n k_i \cdot b_i$ and we set $I_A \equiv \bot$.

Theorem 1. *The interpolants as defined in this section are valid partial tree interpolants for the respective leaf nodes.*

The proof for this theorem is a straight-forward case distinction over the type of leaf node. Details can be found in [14].

Resolution Steps. In a resolution step, we obtain the partial interpolant of the resolvent using the partial interpolants of the premises.

$$\frac{C_1 \vee \ell : I_A^1 \qquad C_2 \vee \neg\ell : I_A^2}{C_1 \vee C_2 : I_A^3}$$

As the first step, we follow McMillan's algorithm and combine the interpolants of the premises either with $\vee$ or with $\wedge$ depending on whether the pivot literal is A or A^c-local. For tree interpolants, this is done separately for each node of the tree interpolation problem, and a literal is seen as A-local if its colour is one of the leaves in the subtree of the node.

$$I_A^3 \equiv \begin{cases} I_A^1 \vee I_A^2 & \text{if } \mathit{colour}(\ell) \in A \\ I_A^1 \wedge I_A^2 & \text{if } \mathit{colour}(\ell) \notin A \end{cases}$$

The formula I_A^3 computed above may still contain variables supported by the antecedents that are no longer supported by the resolvent $C_1 \vee C_2$. Each of those *unsupported* variables must either be replaced by its definition or bound by a quantifier in the partial tree interpolant. More precisely, let v_t be an unsupported variable such that t is not a subterm of t' with $v_{t'} \in \mathit{FreeVars}(I_A^3)$. This variable must always exist, as there is always an outermost unsupported variable. Let $t = f(t_1, \ldots, t_n)$. We replace I_A^3 as follows:

$$I_A^3 \equiv \begin{cases} \exists x.\, I_A^3\{v_t \mapsto x\} & \text{if } f \text{ is } A\text{-local, i.e., } \mathit{partitions}(f) \subseteq A, \\ \forall x.\, I_A^3\{v_t \mapsto x\} & \text{if } f \text{ is } A^c\text{-local, i.e., } \mathit{partitions}(f) \cap A = \emptyset, \\ I_A^3\{v_t \mapsto f(v_{t_1}, \ldots, v_{t_n})\} & \text{if } f \text{ is shared (otherwise).} \end{cases}$$

We do this repeatedly for all variables in $\mathit{FreeVars}(I_A^3)$ that are unsupported. The variables may be treated in any order that respects the partial order induced by the subterm relation as described above. However, all interpolants of the tree interpolant must use the same order.

Theorem 2. *If I_A^1 is a partial tree interpolant of $C_1 \vee \ell$ and I_A^2 is a partial tree interpolant of $C_2 \vee \neg\ell$, then I_A^3 as computed above, after the removal of unsupported variables, is a partial tree interpolant of $C_1 \vee C_2$.*

The proof for this theorem is given in [14].

Example 4 (Resolution). Take the running example and suppose $\ell \equiv g(h(b)) = b$ is the pivot, $I_{\{1\}}^1 \equiv v_{g(h(b))} \leq v_b$ and $I_{\{1\}}^2 \equiv \top$. The interpolants are combined as $I_{\{1\}}^1 \wedge I_{\{1\}}^2$ since $\mathit{colour}(\ell) \notin \{1\}$. This results in the interpolant $v_{g(h(b))} \leq v_b$. After the resolution step, we assume that $v_{g(h(b))}, v_{h(b)}, v_b$ are no longer supported. The outermost variable is $v_{g(h(b))}$, which must be replaced by its definition: $g(v_{h(b)}) \leq v_b$. Now $v_{h(b)}$ is bound by a quantifier, and since h only occurs in partition 1, an existential quantifier is used: $\exists y.\, g(y) \leq v_b$. In the final step, v_b is bound by a universal quantifier since b does not occur in 1, yielding $\forall x. \exists y.\, g(y) \leq x$.

Note that the order of eliminating variables is important. If v_b had been chosen in the first step despite occurring in $\mathit{FlatEQ}(g(h(b)))$, the resulting formula would have been $\exists y.\forall x.g(y) \leq x$. This formula is not logically equivalent and is indeed not a valid interpolant, as it does not follow from $\forall x.g(h(x)) \leq x$.

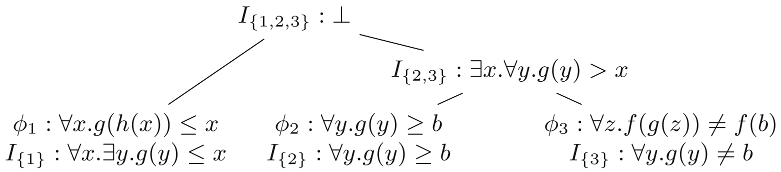

$I_{\{1,2,3\}} : \bot$
$I_{\{2,3\}} : \exists x.\forall y.g(y) > x$
$\phi_1 : \forall x.g(h(x)) \leq x$
$I_{\{1\}} : \forall x.\exists y.g(y) \leq x$
$\phi_2 : \forall y.g(y) \geq b$
$I_{\{2\}} : \forall y.g(y) \geq b$
$\phi_3 : \forall z.f(g(z)) \neq f(b)$
$I_{\{3\}} : \forall y.g(y) \neq b$

Fig. 1. Tree interpolation problem from Example 1 annotated by tree interpolants.

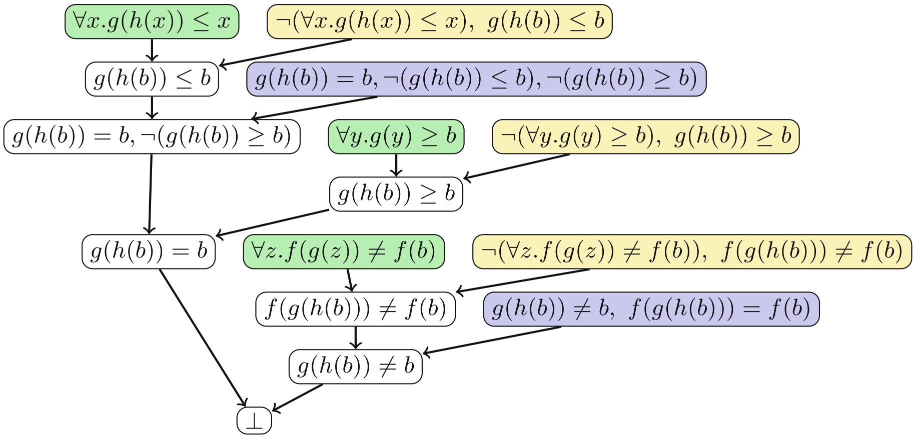

$\forall x.g(h(x)) \leq x$
$\neg(\forall x.g(h(x)) \leq x),\ g(h(b)) \leq b$
$g(h(b)) \leq b$
$g(h(b)) = b, \neg(g(h(b)) \leq b), \neg(g(h(b)) \geq b)$
$g(h(b)) = b, \neg(g(h(b)) \geq b)$
$\forall y.g(y) \geq b$
$\neg(\forall y.g(y) \geq b),\ g(h(b)) \geq b$
$g(h(b)) \geq b$
$g(h(b)) = b$
$\forall z.f(g(z)) \neq f(b)$
$\neg(\forall z.f(g(z)) \neq f(b)),\ f(g(h(b))) \neq f(b)$
$f(g(h(b))) \neq f(b)$
$g(h(b)) \neq b,\ f(g(h(b))) = f(b)$
$g(h(b)) \neq b$
$\bot$

Fig. 2. Resolution proof for Example 1 with input clauses, instantiation lemmas, theory lemmas, and resolvents.

Theorem 3. *The algorithm in this section computes valid tree interpolants from a proof of unsatisfiability.*

Proof. By induction, every node in the proof tree is labelled by a valid partial tree interpolant: Theorem 1 is the base case and Theorem 2 the inductive step. The proof of unsatisfiability ends with the empty clause and its partial interpolant is a tree interpolant for the original problem.

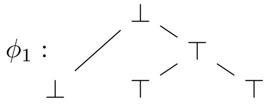
5.2 Full Interpolation Example

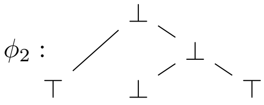
We illustrate the algorithm on our running example (Example 1). Consider the tree interpolation problem given in Fig. 1. The symbol b occurs in partitions 2 and 3, f in 3, g in 1, 2, and 3, and h in 1. Our goal is to compute tree interpolants $I_{\{1\}}$, $I_{\{2\}}$, and $I_{\{3\}}$ for the leaf nodes such that ϕ_1 implies $I_{\{1\}}$, ϕ_2 implies $I_{\{2\}}$, and ϕ_3 implies $I_{\{3\}}$, and tree interpolant $I_{\{2,3\}}$ such that $I_{\{2,3\}}$ is implied by $I_{\{2\}} \wedge I_{\{3\}}$, and $I_{\{1\}} \wedge I_{\{2,3\}}$ implies $\bot$.

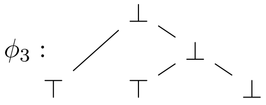
Figure 2 shows an instantiation-based resolution proof for the unsatisfiability of $\phi_1 \wedge \phi_2 \wedge \phi_3$. First, we assign each literal occurring in the proof tree to exactly one partition. We colour each proxy literal for a quantified formula by a partition in which it occurs, e.g., $\mathit{colour}(\forall x. g(h(x)) \leq x) = 1$. For the other literals, we can choose arbitrary colours. We assign the literals $g(h(b)) = b$, $g(h(b)) \leq b$, and $g(h(b)) \geq b$ to partition 2, and the literal $f(g(h(b))) \neq f(b)$ to partition 3. We then compute for each literal ℓ the projection onto each partition, i.e., $\ell \downharpoonleft p_i$. For $\ell \equiv g(h(b)) \leq b$ assigned to partition 2, the projections are given in Example 3. As $g(h(b)) \geq b$ and $g(h(b)) = b$ are assigned to the same partition as ℓ and only differ in the comparison operator, their projections only differ in the comparison operator of the flattened version of the original literal. For the remaining literal $f(g(h(b))) = f(b)$, we get the following projections:

$$\begin{aligned}
f(g(h(b))) = f(b) \downharpoonleft 1 &\equiv v_{g(h(b))} = g(v_{h(b)}) \wedge v_{h(b)} = h(v_b)\\
f(g(h(b))) = f(b) \downharpoonleft 2 &\equiv v_{g(h(b))} = g(v_{h(b)}) \wedge v_b = b\\
f(g(h(b))) = f(b) \downharpoonleft 3 &\equiv v_{f(g(h(b)))} = v_{f(b)} \wedge v_{f(g(h(b)))} = f(v_{g(h(b))}) \wedge\\
& v_{g(h(b))} = g(v_{h(b)}) \wedge v_{f(b)} = f(v_b) \wedge v_b = b
\end{aligned}$$

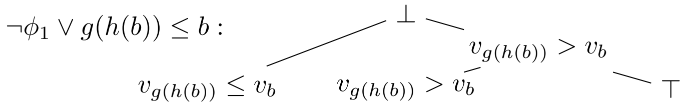
We now compute partial tree interpolants for each node in the proof tree. The first input clause $C \equiv \phi_1$ on the top left of the proof tree is from partition 1. The partial interpolants $I_{\{1\}}$ and $I_{\{1,2,3\}}$ are set to $\neg(\neg C \downharpoonleft^- A^c) \equiv \bot$, and $I_{\{2\}}$, $I_{\{3\}}$, and $I_{\{2,3\}}$ are set to $\neg C \downharpoonleft^- A \equiv \top$. For the input clauses ϕ_2 and ϕ_3, the interpolants are computed analogously. To summarise:

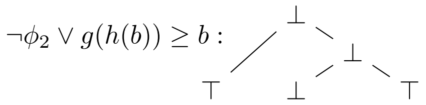


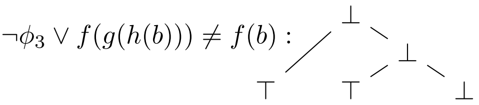




We now compute the partial tree interpolants for the instantiation lemma on the top right of the proof tree. Similar as for the input clauses, we set $I_{\{1\}}$ to $\neg(\neg C \downharpoonleft^- A^c)$, i.e., to $\neg(\neg C \downharpoonleft^- 2) \wedge \neg(\neg C \downharpoonleft^- 3) \equiv v_{g(h(b))} \leq v_b$. Analogously, we compute all other partial tree interpolants for the three instantiation lemmas:



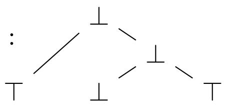




For the trichotomy lemma, the partial tree interpolants can be set to $\neg C \downharpoonleft^- A$ or $\neg(\neg C \downharpoonleft^- A^c)$. Due to our colouring, all literals in the lemma are either in A

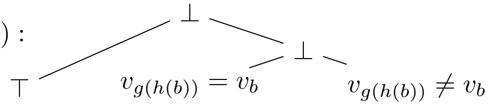
or in A^c. To get the most simple partial interpolants, we set $I_{\{1\}}$ and $I_{\{3\}}$ to $\neg C \downharpoonright^- A \equiv \top$, and $I_{\{2\}}$ and $I_{\{2,3\}}$ to $\neg(\neg C \downharpoonright^- A^c) \equiv \bot$:

$g(h(b)) = b \vee \neg(g(h(b)) \leq b) \vee \neg(g(h(b)) \geq b)$:



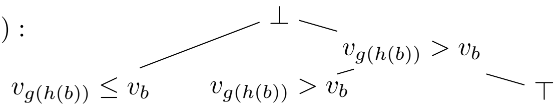
For the congruence lemma, we have $p_f = 3$. The partial tree interpolants $I_{\{1\}}$ and $I_{\{2\}}$ are set to $\neg C \downharpoonright^- A$ as $p_f \notin A$ for these partitions. We get $I_{\{1\}} \equiv \top$ (neither of the flattened literals in $\neg C$ is contained in the projection kernel) and $I_{\{2\}} \equiv v_{g(h(b))} = v_b$, since we chose 2 as the colour of this literal. Similarly, $I_{\{3\}}$ and $I_{\{2,3\}}$ are set to $\neg(\neg C \downharpoonright^- A^c)$. We get $I_{\{3\}} \equiv v_{g(h(b))} \neq v_b$ and $I_{\{2,3\}} \equiv \bot$:

$g(h(b)) \neq b \vee f(g(h(b))) = f(b)$:

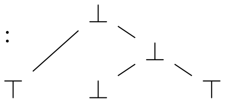


Having computed the partial tree interpolants for all leaves in the proof tree, we now compute the partial tree interpolants for each resolvent. If the colour of the pivot literal ℓ is in the A-part, i.e., $colour(\ell) \in A$, the partial tree interpolant of the resolvent is the disjunction of the partial tree interpolants of its antecedents. Otherwise, if $colour(\ell) \in A^c$, we build the conjunction of the partial tree interpolants of its antecedents. In the resolution step for the resolvent clause $C_3 \equiv g(h(b)) \leq b$, the pivot literal is assigned to partition 1, i.e., $colour(\forall x.g(h(x)) \leq x) = 1$. To obtain $I_{\{1\}}$, we hence build the disjunction of the partial interpolants of the antecedents $C_1 \equiv \forall x.g(h(x)) \leq x$ and $C_2 \equiv \neg(\forall x.g(h(x)) \leq x) \vee\ g(h(b)) \leq b$, so we get $I_{\{1\}} \equiv I^1_{\{1\}} \vee I^2_{\{1\}} \equiv v_{g(h(b))} \leq v_b$. Similarly, we obtain $I_{\{2\}}$, $I_{\{3\}}$ and $I_{\{2,3\}}$ by conjoining the respective partial interpolants. Since the top-left interpolant is only $\top$ or $\bot$ and the colouring of the pivot literal ensures that we either build the conjunction with $\top$ or the disjunction with $\bot$, the resulting tree interpolant of the resolvent is the same as for the top-right clause. The variables $v_{g(h(b))}$ and v_b are both supported by C_3 and thus allowed to appear in the partial interpolant. The resolution steps of the other inner nodes are very similar in that their partial interpolants always equal the partial interpolant of one of their antecedents. To summarise:

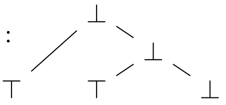
$g(h(b)) \leq b$:
$g(h(b)) = b \vee \neg(g(h(b)) \geq b)$:
$g(h(b)) = b$:



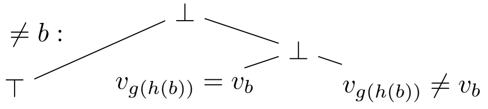
$g(h(b)) \geq b$:



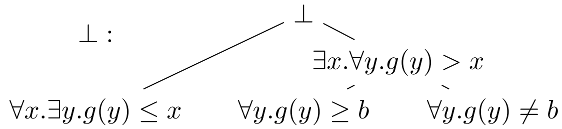
$f(g(h(b))) \neq f(b)$:



$g(h(b)) \neq b$:



The last resolution step is a bit more involved. We have already computed the tree interpolant for partition 1 in Example 4 as $I_{\{1\}} \equiv \forall x.\exists y.g(y) \leq x$. For partition 2, the disjunction $v_{g(h(b))} > v_b \vee v_{g(h(b))} = v_b$ can be simplified to $v_{g(h(b))} \geq v_b$. The outermost variable $v_{g(h(b))}$ is then replaced by $g(v_{h(b)})$, since g occurs in 1 and 2. Then for $v_{h(b)}$ a universal quantifier is introduced, since h only occurs in partition 1, resulting in $\forall y.g(y) \geq v_b$. Finally, v_b is replaced by b, since it occurs in both 2 and 3. This results in $I_{\{2\}} \equiv \forall y.g(y) \geq b$. We omit the computation of the partial interpolant for partitions 3 and the node 23. The partial tree interpolant computed in this step is the tree interpolant of the full interpolation problem:


$\perp$:
$\perp$
$\exists x.\forall y.g(y) > x$
$\forall x.\exists y.g(y) \leq x$
$\forall y.g(y) \geq b$
$\forall y.g(y) \neq b$

6 Combination with Equality-Interpolating Theories

In Sects. 4 and 5, we assign each literal to exactly one partition, such that we can apply McMillan's algorithm to combine partial interpolants of the antecedents to obtain a partial interpolant for the resolvent. In the presence of equality-interpolating theories [25], we can also allow for *mixed* literals where only outermost terms must be assigned to one partition. More precisely, we can allow for equalities $t_1 = t_2$ where the left-hand side t_1 is in one partition and the right-hand side t_2 in another, or linear constraints of the form $c_1 \cdot t_1 + \ldots + c_n \cdot t_n \diamond c_0$ with constants c_i and $\diamond \in \{=, \leq, <, \geq, >\}$, where each t_i is assigned to one partition. Such literals can be treated by applying proof tree preserving tree interpolation [5].

A mixed literal $\ell \equiv t_1 = t_2$ is coloured with two colours p_1 and p_2, so that each colour can be chosen to contain the outermost symbols of t_1 and t_2, respectively. The projections are $\ell \downharpoonleft^- p_1 \equiv v_{t_1} = v_\ell$, $\ell \downharpoonleft^- p_2 \equiv v_\ell = v_{t_2}$ and for the negated literal $\neg\ell \downharpoonleft^- p_1 \equiv EQ_1(v_\ell, v_{t_1})$ and $\neg\ell \downharpoonleft^- p_2 \equiv EQ_2(v_\ell, v_{t_2})$, where v_ℓ is a fresh variable and EQ_1, EQ_2 are shared uninterpreted predicates with $\forall x, y.\neg(EQ_1(x, y) \wedge EQ_2(x, y))$, that are only used for the interpolation algorithm. The partial interpolants for a lemma containing mixed literals will contain the auxiliary variable v_ℓ. If a negated mixed equality occurs in the conflict (the negated lemma), we further require that v_ℓ occurs only in literals of the form $EQ_i(v_\ell, s)$ for some shared term s. Valid interpolants will naturally have this shape, as the interpolated conflict also contains v_ℓ only as first parameter of an EQ_i. We then introduce a new combination rule in the first part of interpolating resolution steps: For a mixed literal ℓ, the two interpolants $I^1[EQ_i(v_\ell, s)]$ and $I^2(v_\ell)$ are combined to $I^1[I^2(s)]$, i.e., interpolant $I^2(s)$ replaces the EQ-literals occurring in the interpolant I^1 to form the resolvent interpolant. This eliminates the variable v_ℓ without introducing a quantifier. The remaining part is unchanged, i.e., we still introduce quantifiers for unsupported flattening variables. A proof that the first step produces a valid resolvent interpolant can be found in [5]. This method produces quantifier-free interpolants if the input formulas were quantifier-free. An example for this method can be found in [13].

7 Implementation in SMTInterpol

We implemented the algorithm in SMTInterpol[2] [6] with a few alterations. First, we used the combination with equality-interpolating theories described in the previous section. Second, we do not apply flattening explicitly. Instead of using an auxiliary variable, the interpolation algorithms for the lemmas include the corresponding term directly. This may result in an interpolant where the interpolant has symbols that are not allowed, because the auxiliary variable was shared but its corresponding function symbol is local to one partition. Only in that case, we introduce the fresh variables for these subterms and replace the offending subterm in the interpolant with its variable. This creates the same interpolants as our presented algorithm, because the latter replaces each variable that stands for a shared function symbol by its definition in the end.

SMTInterpol also supports literals that are shared. If this is done naïvely, the computed interpolants may violate the tree inductivity property (third property in Definition 1). We solve this by treating each literal as occurring in one designated partition when interpolating a lemma (minimizing the number of alternating chains in transitivity lemmas). We then apply Pudlák's resolution rule [21] that has a case for shared literals. Our implementation colours input literals with all partitions it occurs in. For new terms created in the proof, the colour that matches the most outermost function symbols is chosen. If the term uses only symbols from one partition, then it is coloured with that partition. Equalities and inequalities between terms of different partitions are handled with the equality-interpolating procedure to avoid introducing quantifiers when it is not necessary.

8 Conclusion

We presented a tree interpolation algorithm for SMT formulas with quantifiers. The key idea is to virtually flatten each conflict corresponding to a clause in the resolution proof, such that the literals in the flattened version are non-mixed and can be assigned to the different partitions. The colouring of the original literals can even be chosen arbitrarily. Depending on the assigned colours, partial interpolants may contain flattening variables that bridge different partitions, which eventually must be bound by quantifiers.

Our algorithm computes tree interpolants from a single, non-local proof of unsatisfiability obtained independently of the partitioning of the interpolation problem. It supports quantifiers and arbitrary SMT theories, given that the

[2] Official webpage: https://ultimate.informatik.uni-freiburg.de/smtinterpol/
Code available under LGPLv3 at https://github.com/ultimate-pa/smtinterpol.

theory itself supports tree interpolation for its lemmas, and we provided the algorithms for the theory of equality and the theory of linear rational arithmetic.

Correctness proofs for our algorithm are available in [14]. The algorithm is implemented in the open-source SMT solver SMTInterpol.

References

1. Blanc, R., Gupta, A., Kovács, L., Kragl, B.: Tree interpolation in vampire. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 173–181. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_13
2. Bonacina, M.P., Johansson, M.: On interpolation in automated theorem proving. J. Autom. Reason. **54**(1), 69–97 (2015)
3. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_28
4. Christ, J., Hoenicke, J.: Instantiation-based interpolation for quantified formulae. In: Decision Procedures in Software, Hardware and Bioware. Dagstuhl Seminar Proceedings, vol. 10161. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2010)
5. Christ, J., Hoenicke, J.: Proof tree preserving tree interpolation. J. Autom. Reasoning **57**(1), 67–95 (2016)
6. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: an interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31759-0_19
7. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symb. Log. **22**(3), 269–285 (1957)
8. Dillig, I., Dillig, T., Aiken, A.: Cuts from proofs: a complete and practical technique for solving linear inequalities over integers. Formal Methods Syst. Des. **39**(3), 246–260 (2011)
9. Gleiss, B., Kovács, L., Suda, M.: Splitting proofs for interpolation. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 291–309. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_18
10. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: POPL, pp. 331–344. ACM (2011)
11. Gurfinkel, A., Rollini, S.F., Sharygina, N.: Interpolation properties and SAT-based model checking. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 255–271. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_19
12. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: POPL, pp. 471–482. ACM (2010)
13. Henkel, E., Hoenicke, J., Schindler, T.: Proof tree preserving sequence interpolation of quantified formulas in the theory of equality. In: SMT. CEUR Workshop Proceedings, vol. 2908, pp. 3–16. CEUR-WS.org (2021)
14. Henkel, E., Hoenicke, J., Schindler, T.: Choose your colour: tree interpolation for quantified formulas in SMT. CoRR abs/2305.11667 (2023). https://arxiv.org/abs/2305.11667
15. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, pp. 232–244. ACM (2004)

16. Hoenicke, J., Majumdar, R., Podelski, A.: Thread modularity at many levels: a pearl in compositional verification. In: POPL, pp. 473–485. ACM (2017)
17. Kovács, L., Voronkov, A.: First-order interpolation and interpolating proof systems. In: LPAR. EPiC Series in Computing, vol. 46, pp. 49–64. EasyChair (2017)
18. McMillan, K.L.: An interpolating theorem prover. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 16–30. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_2
19. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_31
20. Monniaux, D., Gonnord, L.: Cell morphing: from array programs to array-free horn clauses. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 361–382. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_18
21. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. J. Symb. Log. **62**(3), 981–998 (1997)
22. Rollini, S.F., Bruttomesso, R., Sharygina, N.: An efficient and flexible approach to resolution proof reduction. In: Barner, S., Harris, I., Kroening, D., Raz, O. (eds.) HVC 2010. LNCS, vol. 6504, pp. 182–196. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19583-9_17
23. Rümmer, P., Hojjat, H., Kuncak, V.: On recursion-free horn clauses and Craig interpolation. Formal Methods Syst. Des. **47**(1), 1–25 (2015)
24. Seghir, M.N., Podelski, A., Wies, T.: Abstraction refinement for quantified array assertions. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 3–18. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03237-0_3
25. Yorsh, G., Musuvathi, M.: A combination method for generating interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 353–368. Springer, Heidelberg (2005). https://doi.org/10.1007/11532231_26

Proving Termination of C Programs with Lists

Jera Hensel(✉) and Jürgen Giesl(✉)

LuFG Informatik 2, RWTH Aachen University, Aachen, Germany
{hensel,giesl}@informatik.rwth-aachen.de

Abstract. There are many techniques and tools to prove termination of C programs, but up to now these tools were not very powerful for fully automated termination proofs of programs whose termination depends on recursive data structures like lists. We present the first approach that extends powerful techniques for termination analysis of C programs (with memory allocation and explicit pointer arithmetic) to lists.

1 Introduction

In [11,16,17,25], we introduced an approach for automatic termination analysis of C that also handles programs whose termination relies on the relation between allocated memory addresses and the data stored at such addresses. This approach is implemented in our tool AProVE [14]. Instead of analyzing C directly, AProVE compiles the program to LLVM code using Clang [9]. Then it constructs a (finite) symbolic execution graph (SEG) such that every program run corresponds to a path through the SEG. AProVE proves memory safety during the construction of the SEG to ensure absence of undefined behavior (which would also allow non-termination). Afterwards, the SEG is transformed into an integer transition system (ITS) such that all paths through the SEG (and hence, the C program) are terminating if the ITS is terminating. To analyze termination of the ITS, AProVE applies standard techniques and calls the tools T2 [7] and LoAT [12,13] to detect non-termination of ITSs. However, like other termination tools for C, up to now AProVE supported dynamic data structures only in a very restricted way.

```
struct list {
  unsigned int value;
  struct list* next;   };

int main() {
  // initialize length
  unsigned int n = nondet_uint();
  // initialize list of length n
  struct list* tail = NULL;
  struct list* curr;
  for (unsigned int k = 0; k < n; k++) {
    curr = malloc(sizeof(struct list));
    curr->value = nondet_uint();
    curr->next = tail;
    tail = curr;                       }
  // traverse list
  struct list* ptr = tail;
  while(ptr != NULL) {
    ptr = *((struct list**)((void*)ptr +
          offsetof(struct list, next)));}}
```

In this paper, we introduce a novel technique to analyze C programs on lists. In the program on the right, nondet_uint returns a random unsigned integer. The for loop creates a list of n random numbers if $\texttt{n} > 0$. The while loop traverses this list via pointer arithmetic: Starting with tail, it computes the address of the next field of the

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2).

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 266–285, 2023.
https://doi.org/10.1007/978-3-031-38499-8_16

current element by adding the offset of the next field within a list to the address of the current list and dereferencing the computed address (i.e., the content of the next field). This is done by offsetof, defined in the C library stddef.h.[1] Since the list is acyclic and the next pointer of its last element is the null pointer, list traversal always terminates. Of course, the while loop could also traverse the list via ptr = ptr->next, but in C, memory accesses can be combined with pointer arithmetic. This example contains both the access via curr->next (when initializing the list) and pointer arithmetic (when traversing the list).

We present a new general technique to infer *list invariants* via symbolic execution, which express all properties that are crucial for memory safety and termination. In our example, the list invariant contains the information that dereferencing the next pointer in the while loop is safe and that one finally reaches the null pointer. In general, our novel list invariants allow us to abstract from detailed information about lists (e.g., about their intermediate elements) such that abstract states with "similar" lists can be merged and generalized during the symbolic execution in order to obtain finite SEGs. At the same time, list invariants express enough information about the lists (e.g., their length, their start address, etc.) such that memory safety and termination can still be proved.

We define the abstract states used for symbolic execution in Sect. 2. In Sect. 3, after recapitulating the construction of SEGs, we adapt our techniques for merging and generalizing states from [25] to infer list invariants. Moreover, we adapt those rules for symbolic execution that are affected by introducing list invariants. Section 4 discusses the generation of ITSs and the soundness of our approach. Section 5 gives an overview on related work. Moreover, we evaluate the implementation of our approach in the tool AProVE using benchmark sets from *SV-COMP* [3] and the *Termination Competition* [15]. All proofs can be found in [18].

Limitations. To ease the presentation, in this paper we treat integer types as unbounded. Moreover, we assume that a program consists of a single non-recursive function and that values may be stored at any address. Our approach can also deal with bitvectors, data alignments, and programs with arbitrary many (possibly recursive) functions, see [11,16,25] for details. However, so far only lists without sharing can be handled by our new technique. Extending it to more general recursive data structures is one of the main challenges for future work.

2 Abstract States for Symbolic Execution

The LLVM code for the for loop is given on the next page. It is equivalent to the code produced by Clang without optimizations on a 64-bit computer. We explain it in detail in Sect. 3. To ease readability, we omitted instructions and keywords that are irrelevant for our presentation, renamed variables, and wrote list

[1] Note that ptr + n increases ptr by n times the size of the type *ptr. As we want to increase ptr by a number of bytes and ptr is not an i8 pointer, we first cast ptr to void*. Then ((void*)ptr + offsetof(struct list, next)) contains the next pointer, so we cast our computed address to struct list** before dereferencing it.

instead of `struct.list`. Moreover, we gave the C instructions (in gray) before the corresponding LLVM code. The code consists of several *basic blocks* including `cmpF` and `bodyF` (corresponding to the loop comparison and body).

```
list = type { i32, list* }

define i32 @main() { ...
cmpF:
  k < n
  0: k = load i32, i32* k_ad
  1: kltn = icmp ult i32 k, n
  2: br i1 kltn, label bodyF, label initPtr
bodyF:
  curr = malloc(sizeof(struct list));
  0: mem = call i8* @malloc(i64 16)
  1: curr = bitcast i8* mem to list*
  curr->value = nondet_uint();
  2: nondet = call i32 @nondet_uint()
  3: curr_val = getelementptr list,
              list* curr, i32 0, i32 0
  4: store i32 nondet, i32* curr_val
  curr->next = tail;
  5: tail = load list*, list** tail_ptr
  6: curr_next = getelementptr list,
               list* curr, i32 0, i32 1
  7: store list* tail, list** curr_next
  tail = curr;
  8: store list* curr, list** tail_ptr
  k++
  9: kinc = add i32 k, 1
  10:store i32 kinc, i32* k_ad
  11:br label cmpF
  ...                                     }
```

We now recapitulate the *abstract states* of [25] used for symbolic execution and extend them by a component *LI* for list invariants, i.e., they have the form $((\mathtt{b}, i), \mathit{LV}, \mathit{AL}, \mathit{PT}, \mathit{LI}, \mathit{KB})$. The first component is a *program position* $(\mathtt{b}, i)$, indicating that instruction i of block $\mathtt{b}$ is executed next. $\mathit{Pos} \subseteq (\mathit{Blks} \times \mathbb{N})$ is the set of all program positions, and *Blks* are all basic blocks.

The second component is a partial injective function $\mathit{LV} : \mathcal{V}_\mathcal{P} \rightharpoonup \mathcal{V}_{sym}$, which maps *local program variables* $\mathcal{V}_\mathcal{P}$ of the program $\mathcal{P}$ to an infinite set $\mathcal{V}_{sym}$ of symbolic variables with $\mathcal{V}_{sym} \cap \mathcal{V}_\mathcal{P} = \varnothing$. We identify *LV* with the set of equations $\{\mathtt{x} = \mathit{LV}(\mathtt{x}) \mid \mathtt{x} \in \mathit{domain}(\mathit{LV})\}$ and we often extend *LV* to a function from $\mathcal{V}_\mathcal{P} \uplus \mathbb{N}$ to $\mathcal{V}_{sym} \uplus \mathbb{N}$ by defining $\mathit{LV}(n) = n$ for all $n \in \mathbb{N}$.

The third component of each state is a set *AL* of (bytewise) allocations $[\![v_1,\, v_2]\!]$ with $v_1, v_2 \in \mathcal{V}_{sym}$, which indicate that $v_1 \leq v_2$ and that all addresses between v_1 and v_2 have been allocated. We require any two entries $[\![v_1,\, v_2]\!]$ and $[\![w_1,\, w_2]\!]$ from *AL* with $v_1 \neq w_1$ or $v_2 \neq w_2$ to be disjoint.

The fourth and fifth components *PT* and *LI* model the memory contents. *PT* contains "points-to" entries of the form $v_1 \hookrightarrow_{\mathtt{ty}} v_2$ where $v_1, v_2 \in \mathcal{V}_{sym}$ and `ty` is an LLVM type, meaning that the address v_1 of type `ty` points to v_2. In contrast, the set *LI* of *list invariants* (which is new compared to [25]) does not describe pointwise memory contents but contains invariants $v_{ad} \overset{v_\ell}{\hookrightarrow}_{\mathtt{ty}} [(\mathit{off}_i : \mathtt{ty}_i : v_i..\hat{v}_i)]_{i=1}^n$ where $n \in \mathbb{N}_{>0}$, $v_{ad}, v_\ell, v_i, \hat{v}_i \in \mathcal{V}_{sym}$, $\mathit{off}_i \in \mathbb{N}$ for all $1 \leq i \leq n$, `ty` and $\mathtt{ty}_i$ are LLVM types for all $1 \leq i \leq n$, and there is exactly one "recursive field" $1 \leq j \leq n$ such that $\mathtt{ty}_j = \mathtt{ty*}$.[2] Such an invariant represents a `struct ty` with n fields that corresponds to a recursively defined list of length v_ℓ. Here, v_{ad} points to the first list element, the i-th field starts at address $v_{ad} + \mathit{off}_i$ (i.e., with offset off_i)[3] and has type $\mathtt{ty}_i$, and the values of the i-th fields of the first and last list element are v_i and $\hat{v}_i$, respectively. For example, the following list invariant (1) represents all lists of length x_ℓ and type `list` whose elements store a 32-bit integer in their first field and the pointer to the next element in their second field

[2] Soundness of our approach is not affected if there are other recursive fields, but our symbolic execution technique for list traversal on list invariants in Sect. 3.2.2 can only be applied if the traversal is done along field j.

[3] The field offsets can be computed using the data layout string in the LLVM program.

with offset 8. The first list element starts at address $x_{\texttt{mem}}$, the second starts at address $x_{\texttt{next}}$, and the last element contains the null pointer. Moreover, the first element stores the integer value $x_{\texttt{nd}}$ and the last list element stores the integer $\hat{x}_{\texttt{nd}}$.

$$x_{\texttt{mem}} \xhookrightarrow{x_\ell}_{\texttt{list}} [(0 : \texttt{i32} : x_{\texttt{nd}}..\hat{x}_{\texttt{nd}}), (8 : \texttt{list*} : x_{\texttt{next}}..0)] \tag{1}$$

For example, this invariant represents the list with the allocation $\llbracket x_{\texttt{mem}},\, x_{\texttt{mem}}{+}15 \rrbracket$, where the first four bytes store the integer 5 and the last eight bytes store the pointer $x_{\texttt{next}}$, and the allocation $\llbracket x_{\texttt{next}},\, x_{\texttt{next}}{+}15 \rrbracket$, where the first four bytes store the integer 2 and the last eight bytes store the null pointer (i.e., the address 0). Here, we have $x_\ell = 2$. Section 3.2.2 will show that the expressiveness of our list invariants is indeed needed to prove termination of programs that traverse a list.

The last component of a state is a *k̲nowledge b̲ase KB* of quantifier-free first-order formulas that express integer arithmetic properties of $\mathcal{V}_{sym}$. We identify *sets* of first-order formulas $\{\varphi_1, \ldots, \varphi_m\}$ with their conjunction $\varphi_1 \wedge \ldots \wedge \varphi_m$.

A special state *ERR* is reached if we cannot prove absence of undefined behavior (e.g., if memory safety might be violated by dereferencing the null pointer).

As an example, the following abstract state (2) represents concrete states at the beginning of the block `cmpF`, where the program variable `curr` is assigned the symbolic variable $x_{\texttt{mem}}$, the allocation $\llbracket x_{\texttt{k\_ad}},\, x_{\texttt{k\_ad}}^{end} \rrbracket$ consisting of 4 bytes stores the value $x_{\texttt{kinc}}$, and $x_{\texttt{mem}}$ points to the first element of a list of length x_ℓ (equal to $x_{\texttt{kinc}}$) that satisfies the list invariant (1). (This state will later be obtained during the symbolic execution, see State O in Fig. 3 in Sect. 3.1.)

$$\boxed{\begin{array}{l}(\texttt{cmpF}, 0),\ \{\texttt{curr} = x_{\texttt{mem}},\ \texttt{kinc} = x_{\texttt{kinc}},\ \ldots\},\ \{\llbracket x_{\texttt{k\_ad}},\ x_{\texttt{k\_ad}}^{end} \rrbracket,\ \ldots\},\ \{x_{\texttt{k\_ad}} \hookrightarrow_{\texttt{i32}} x_{\texttt{kinc}},\ \ldots\},\\ \{x_{\texttt{mem}} \xhookrightarrow{x_\ell}_{\texttt{list}} [(0 : \texttt{i32} : x_{\texttt{nd}}..\hat{x}_{\texttt{nd}}), (8 : \texttt{list*} : x_{\texttt{next}}..0)]\},\ \{x_{\texttt{k\_ad}}^{end} = x_{\texttt{k\_ad}} + 3,\ x_\ell = x_{\texttt{kinc}},\ \ldots\}\end{array}} \tag{2}$$

A state $s = (p, LV, AL, PT, LI, KB)$ is *represented by a formula* $\langle s \rangle$ which contains KB and encodes AL, PT, and LI in first-order logic. This allows us to use standard SMT solving for all reasoning during the construction of the SEG. Moreover, $\langle s \rangle$ is also used for the generation of the ITS afterwards. The encoding of AL and PT is as in [25], see [18]: $\langle s \rangle$ contains formulas which express that allocated addresses are positive, that allocations represent disjoint memory areas, that equal addresses point to equal values, and that addresses are different if they point to different values. For each element of LI, we add the following new formulas to $\langle s \rangle$ which express that the list length v_ℓ is ≥ 1 and the address v_{ad} of the first element is not null. If $v_\ell = 1$, then the values v_i and $\hat{v}_i$ of the fields in the first and the last element are equal. If $v_\ell \geq 2$, then the `next` pointer v_j in the first element must not be null. Finally, if there is a field whose values v_k and $\hat{v}_k$ differ in the first and the last element, then the length v_ℓ must be ≥ 2.

$$\begin{array}{l}\{v_\ell \geq 1 \wedge v_{ad} \geq 1 \mid (v_{ad} \xhookrightarrow{v_\ell}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : v_i..\hat{v}_i)]_{i=1}^n) \in LI\} \cup \\ \{\bigwedge_{i=1}^n v_i = \hat{v}_i \mid (v_{ad} \xhookrightarrow{v_\ell}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : v_i..\hat{v}_i)]_{i=1}^n) \in LI \text{ and } \models \langle s \rangle \Rightarrow v_\ell = 1\} \cup \\ \{v_j \geq 1 \mid (v_{ad} \xhookrightarrow{v_\ell}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : v_i..\hat{v}_i)]_{i=1}^n) \in LI \text{ with } \texttt{ty}_j = \texttt{ty}* \text{ and } \models \langle s \rangle \Rightarrow v_\ell \geq 2\} \cup \\ \{v_\ell \geq 2 \mid (v_{ad} \xhookrightarrow{v_\ell}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : v_i..\hat{v}_i)]_{i=1}^n) \in LI \text{ and } \exists\, k \in \mathbb{N}_{>0}, k \leq n, \text{ s.t. } \models \langle s \rangle \Rightarrow v_k \neq \hat{v}_k\}\end{array}$$

In *concrete* states c, all values of variables and memory contents are determined uniquely. To ease the formalization, we assume that all integers are

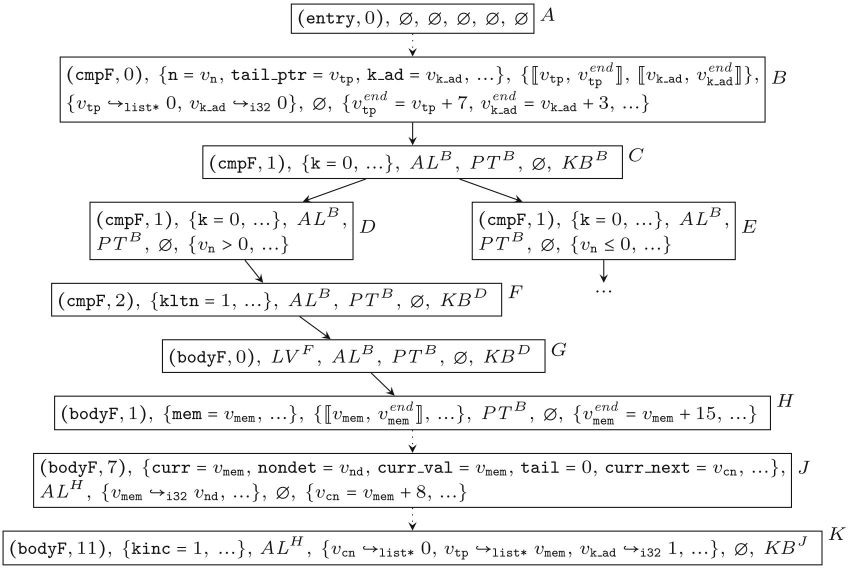

(entry, 0), ∅, ∅, ∅, ∅, ∅ A
(cmpF, 0), {n = v_n, tail_ptr = v_tp, k_ad = v_k_ad, ...}, {⟦v_tp, v_tp^end⟧, ⟦v_k_ad, v_k_ad^end⟧}, {v_tp ↪list* 0, v_k_ad ↪i32 0}, ∅, {v_tp^end = v_tp + 7, v_k_ad^end = v_k_ad + 3, ...} B
(cmpF, 1), {k = 0, ...}, AL^B, PT^B, ∅, KB^B C
(cmpF, 1), {k = 0, ...}, AL^B, PT^B, ∅, {v_n > 0, ...} D
(cmpF, 1), {k = 0, ...}, AL^B, PT^B, ∅, {v_n ≤ 0, ...} E
...
(cmpF, 2), {kltn = 1, ...}, AL^B, PT^B, ∅, KB^D F
(bodyF, 0), LV^F, AL^B, PT^B, ∅, KB^D G
(bodyF, 1), {mem = v_mem, ...}, {⟦v_mem, v_mem^end⟧, ...}, PT^B, ∅, {v_mem^end = v_mem + 15, ...} H
(bodyF, 7), {curr = v_mem, nondet = v_nd, curr_val = v_mem, tail = 0, curr_next = v_cn, ...}, AL^H, {v_mem ↪i32 v_nd, ...}, ∅, {v_cn = v_mem + 8, ...} J
(bodyF, 11), {kinc = 1, ...}, AL^H, {v_cn ↪list* 0, v_tp ↪list* v_mem, v_k_ad ↪i32 1, ...}, ∅, KB^J K

Fig. 1. SEG for the First Iteration of the `for` Loop

unsigned and refer to [16] for the general case. So for all $v \in \mathcal{V}_{sym}(c)$ (i.e., all $v \in \mathcal{V}_{sym}$ occurring in c) we have $\models \langle c \rangle \Rightarrow v = n$ for some $n \in \mathbb{N}$. Moreover, here PT only contains information about allocated addresses and $LI = \varnothing$ since the abstract knowledge in list invariants is unnecessary if all memory contents are known.

For instance, all concrete states $((\texttt{cmpF}, 0), LV, AL, PT, \varnothing, KB)$ represented by the state (2) contain ℓ allocations of 16 bytes for some $\ell \geq 1$, where in the first four bytes a 32-bit integer is stored and in the last eight bytes the address of the next allocation (or 0, in case of the last allocation) is stored.

See [18] for a formal definition to determine which concrete states are represented by a state s. To this end, as in [25] we define a *separation logic* formula $\langle s \rangle_{SL}$ which also encodes the knowledge contained in the memory components of states. To extend this formula to list invariants, we use a fragment similar to *quantitative* separation logic [4], extending conventional separation logic by list predicates. For any state s, we have $\models \langle s \rangle_{SL} \Rightarrow \langle s \rangle$, i.e., $\langle s \rangle$ is a weakened version of $\langle s \rangle_{SL}$ that we use for symbolic execution and the termination proof.

3 Symbolic Execution with List Invariants

We first recapitulate the construction of SEGs. Then, Sect. 3.1 extends the technique for *merging* and generalization of states from [25] to infer list invariants. Finally, we adapt the rules for symbolic execution to list invariants in Sect. 3.2.

Our symbolic execution starts with a state A at the first instruction of the first block (called `entry` in our example). Figure 1 shows the first iteration of the for loop. Dotted arrows indicate that we omitted some symbolic execution steps. For every state, we perform symbolic execution by applying the corresponding inference rule as in [25] to compute its successor state(s) and repeat this until all paths end in return states. We call an SEG with this property *complete*.

As an example, we recapitulate the inference rule for the load instruction in the case where a value is loaded from allocated and initialized memory. It loads the value of type ty that is stored at the address ad to the program variable x. Let $\mathit{size}(\texttt{ty})$ denote the size of ty in bytes for any LLVM type ty. If we can prove that there is an allocation $[\![v_1,\, v_2]\!]$ containing all addresses $LV(\texttt{ad}), \ldots, LV(\texttt{ad}) + \mathit{size}(\texttt{ty}) - 1$ and there exists an entry $(w_1 \hookrightarrow_{\texttt{ty}} w_2) \in PT$ such that w_1 is equal to the address $LV(\texttt{ad})$ loaded from, then we transform the state s at position $p = (\texttt{b}, i)$ to a state s' at position $p^+ = (\texttt{b}, i+1)$. In s', a fresh symbolic variable w is assigned to x and $w = w_2$ is added to KB. We write $LV[\texttt{x} := w]$ for the function where $LV[\texttt{x} := w](\texttt{x}) = w$ and $LV[\texttt{x} := w](\texttt{y}) = LV(\texttt{y})$ for all $\texttt{y} \neq \texttt{x}$.

load **from initialized allocated memory** (p: "x = load ty, ty* ad", $\texttt{x}, \texttt{ad} \in \mathcal{V}_{\mathcal{P}}$)

$$\frac{s = (p,\ LV,\ AL,\ PT,\ LI,\ KB)}{s' = (p^+,\ LV[\texttt{x} := w],\ AL,\ PT,\ LI,\ KB \cup \{w = w_2\})} \quad \text{if } w \in \mathcal{V}_{sym} \text{ is fresh and}$$

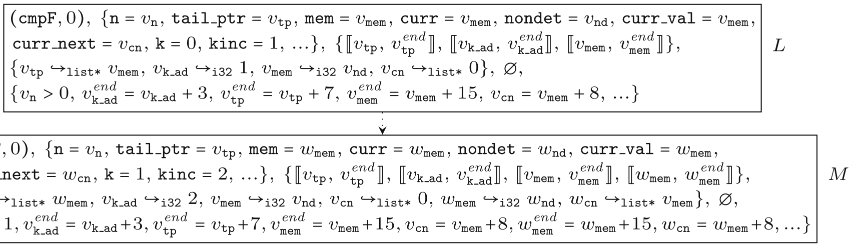
- there is $[\![v_1,\, v_2]\!] \in AL$ with $\models \langle s \rangle \Rightarrow (v_1 \leq LV(\texttt{ad}) \wedge LV(\texttt{ad}) + \mathit{size}(\texttt{ty}) - 1 \leq v_2)$
- there are $w_1, w_2 \in \mathcal{V}_{sym}$ with $\models \langle s \rangle \Rightarrow (LV(\texttt{ad}) = w_1)$ and $(w_1 \hookrightarrow_{\texttt{ty}} w_2) \in PT$

In our example, the entry block comprises the first three lines of the C program and the initialization of the pointer to the loop variable k: First, a non-deterministic unsigned integer is assigned to n, i.e., $(\texttt{n} = v_{\texttt{n}}) \in LV^B$, where $v_{\texttt{n}}$ is not restricted. Moreover, memory for the pointers tail_ptr and k_ad is allocated and they point to tail = NULL and k = 0, respectively ($\texttt{tail\_ptr} = v_{\texttt{tp}}$ and $\texttt{k\_ad} = v_{\texttt{k\_ad}}$ with $(v_{\texttt{tp}} \hookrightarrow_{\texttt{list*}} 0), (v_{\texttt{k\_ad}} \hookrightarrow_{\texttt{i32}} 0) \in PT^B$). For simplicity, in Fig. 1 we use concrete values directly instead of introducing fresh variables for them. Since we assume a 64-bit architecture, tail_ptr's allocation contains 8 bytes. For the integer value of k, only 4 bytes are allocated. Alignments and pointer sizes depend on the memory layout and are given in the LLVM program.

State C results from B by evaluating the load instruction at $(\texttt{cmpF}, 0)$, see the above load rule. Thus, there is an *evaluation edge* from B to C.

The next instruction is an integer comparison whose Boolean return value depends on whether the unsigned value of k is less than the one of n. If we cannot decide the validity of a comparison, we refine the state into two successor states. Thus, the states D and E (with $(v_{\texttt{n}} > 0) \in KB^D$ and $(v_{\texttt{n}} \leq 0) \in KB^E$) are reached by *refinement edges* from State C. Evaluating D yields $\texttt{kltn} = 1$ in F. Therefore, the branch instruction leads to the block bodyF in State G. State E is evaluated to a state with $\texttt{kltn} = 0$. This path branches to the block initPtr and terminates quickly as tail_ptr points to an empty list.

The instruction at $(\texttt{bodyF}, 0)$ allocates 16 bytes of memory starting at $v_{\texttt{mem}}$ in State H. The next instruction casts the pointer to the allocation from i8* to list* and assigns it to curr. Now the allocated area can be treated as a list element. Then nondet_uint() is invoked to assign a 32-bit integer to nondet.



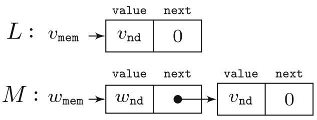
$$(\mathtt{cmpF}, 0),\ \{\mathtt{n} = v_{\mathtt{n}},\ \mathtt{tail\_ptr} = v_{\mathtt{tp}},\ \mathtt{mem} = v_{\mathtt{mem}},\ \mathtt{curr} = v_{\mathtt{mem}},\ \mathtt{nondet} = v_{\mathtt{nd}},\ \mathtt{curr\_val} = v_{\mathtt{mem}},\ \mathtt{curr\_next} = v_{\mathtt{cn}},\ \mathtt{k} = 0,\ \mathtt{kinc} = 1,\ \ldots\},\ \{\llbracket v_{\mathtt{tp}}, v_{\mathtt{tp}}^{end}\rrbracket, \llbracket v_{\mathtt{k\_ad}}, v_{\mathtt{k\_ad}}^{end}\rrbracket, \llbracket v_{\mathtt{mem}}, v_{\mathtt{mem}}^{end}\rrbracket\},\ \{v_{\mathtt{tp}} \hookrightarrow_{\mathtt{list*}} v_{\mathtt{mem}},\ v_{\mathtt{k\_ad}} \hookrightarrow_{\mathtt{i32}} 1,\ v_{\mathtt{mem}} \hookrightarrow_{\mathtt{i32}} v_{\mathtt{nd}},\ v_{\mathtt{cn}} \hookrightarrow_{\mathtt{list*}} 0\},\ \varnothing,\ \{v_{\mathtt{n}} > 0,\ v_{\mathtt{k\_ad}}^{end} = v_{\mathtt{k\_ad}} + 3,\ v_{\mathtt{tp}}^{end} = v_{\mathtt{tp}} + 7,\ v_{\mathtt{mem}}^{end} = v_{\mathtt{mem}} + 15,\ v_{\mathtt{cn}} = v_{\mathtt{mem}} + 8,\ \ldots\}$$

$$(\mathtt{cmpF}, 0),\ \{\mathtt{n} = v_{\mathtt{n}},\ \mathtt{tail\_ptr} = v_{\mathtt{tp}},\ \mathtt{mem} = w_{\mathtt{mem}},\ \mathtt{curr} = w_{\mathtt{mem}},\ \mathtt{nondet} = w_{\mathtt{nd}},\ \mathtt{curr\_val} = w_{\mathtt{mem}},\ \mathtt{curr\_next} = w_{\mathtt{cn}},\ \mathtt{k} = 1,\ \mathtt{kinc} = 2,\ \ldots\},\ \{\llbracket v_{\mathtt{tp}}, v_{\mathtt{tp}}^{end}\rrbracket, \llbracket v_{\mathtt{k\_ad}}, v_{\mathtt{k\_ad}}^{end}\rrbracket, \llbracket v_{\mathtt{mem}}, v_{\mathtt{mem}}^{end}\rrbracket, \llbracket w_{\mathtt{mem}}, w_{\mathtt{mem}}^{end}\rrbracket\},\ \{v_{\mathtt{tp}} \hookrightarrow_{\mathtt{list*}} w_{\mathtt{mem}},\ v_{\mathtt{k\_ad}} \hookrightarrow_{\mathtt{i32}} 2,\ v_{\mathtt{mem}} \hookrightarrow_{\mathtt{i32}} v_{\mathtt{nd}},\ v_{\mathtt{cn}} \hookrightarrow_{\mathtt{list*}} 0,\ w_{\mathtt{mem}} \hookrightarrow_{\mathtt{i32}} w_{\mathtt{nd}},\ w_{\mathtt{cn}} \hookrightarrow_{\mathtt{list*}} v_{\mathtt{mem}}\},\ \varnothing,\ \{v_{\mathtt{n}} > 1, v_{\mathtt{k\_ad}}^{end} = v_{\mathtt{k\_ad}}+3, v_{\mathtt{tp}}^{end} = v_{\mathtt{tp}}+7, v_{\mathtt{mem}}^{end} = v_{\mathtt{mem}}+15, v_{\mathtt{cn}} = v_{\mathtt{mem}}+8, w_{\mathtt{mem}}^{end} = w_{\mathtt{mem}}+15, w_{\mathtt{cn}} = w_{\mathtt{mem}}+8, \ldots\}$$

Fig. 2. Second Iteration of the `for` Loop

The `getelementptr` instruction computes the address of the integer field of the list element by indexing this field (the second `i32 0`) based on the start address (`curr`). The first index (`i32 0`) specifies that a field of `*curr` itself is computed and not of another list stored after `*curr`. Since the address of the integer value of the list element coincides with the start address of the list element, this instruction assigns $v_{\mathtt{mem}}$ to `curr_val`. Afterwards, the value of `nondet` is stored at `curr_val` ($v_{\mathtt{mem}} \hookrightarrow_{\mathtt{i32}} v_{\mathtt{nd}}$), the value `0` stored at $v_{\mathtt{tp}}$ is loaded to `tail`, and a second `getelementptr` instruction computes the address of the recursive field of the current list element ($v_{\mathtt{cn}}{=}v_{\mathtt{mem}}{+}8$) and assigns it to `curr_next`, leading to state J. In the path to K, the values of `tail` and `curr` are stored at `curr_next` and `tail_ptr`, respectively ($v_{\mathtt{cn}} \hookrightarrow_{\mathtt{list*}} 0$, $v_{\mathtt{tp}} \hookrightarrow_{\mathtt{list*}} v_{\mathtt{mem}}$). Finally, the incremented value of `k` is assigned to `kinc` and stored at `k_ad` ($v_{\mathtt{k\_ad}} \hookrightarrow_{\mathtt{i32}} 1$).

To ensure a finite graph construction, when a program position is reached for the second time, we try to merge the states at this position to a *generalized* state. However, this is only meaningful if the domains of the LV functions of the two states coincide (i.e., the states consider the same program variables). Therefore, after the branch from the loop body back to `cmpF` (see State L in Fig. 2), we evaluate the loop a second time and reach M. Here, a second list element with value $w_{\mathtt{nd}}$ and a `next` pointer $w_{\mathtt{cn}}$ pointing to $v_{\mathtt{mem}}$ has been stored at a new allocation $\llbracket w_{\mathtt{mem}}, w_{\mathtt{mem}}^{end}\rrbracket$. Now, `curr` points to the new element and `k` has been incremented again, so `k_ad` points to 2.



3.1 Inferring List Invariants and Generalization of States

As mentioned, our goal is to merge L and M to a more general state O that represents all states which are represented by L or M. The challenging part during generalization is to find loop invariants automatically that always hold at this position and provide sufficient information to prove termination of the loop. For O, we can neither use the information that `curr` points to a struct whose `next` field contains the null pointer (as in L), nor that its `next` field points to another struct whose `next` field contains the null pointer (as in M).

With the approach of [25], when merging states like L and M where a list has different lengths, the merged state would only contain those list elements that are allocated in both states (often this is only the first element). Then elements which are the null pointer in one but not in the other state are lost. Hence, proving memory safety (and thus, also termination) fails when the list is traversed afterwards, since now there might be `next` pointers to non-allocated memory.

We solve this problem by introducing *list invariants*. In our example, we will infer an invariant stating that `curr` points to a list of length $x_\ell \geq 1$. This invariant also implies that all struct fields are allocated and that there is no sharing.

To this end, we adapt the merging heuristic from [25]. To merge two states s and s' at the same program position with $domain(LV^s) = domain(LV^{s'})$, we introduce a fresh symbolic variable $x_{\mathtt{var}}$ for each program variable `var` and use instantiations μ_s and $\mu_{s'}$ which map $x_{\mathtt{var}}$ to the corresponding symbolic variables of s and s'. For the merged state $\overline{s}$, we set $LV^{\overline{s}}(\mathtt{var}) = x_{\mathtt{var}}$. Moreover, we identify corresponding variables that only occur in the memory components and extend μ_s and $\mu_{s'}$ accordingly. In a second step, we check which constraints from the memory components and the knowledge base hold in both states in order to find invariants that we can add to the memory components and the knowledge base of $\overline{s}$. For example, if $[\![\mu_s(x),\, \mu_s(x^{end})]\!] \in AL^s$ and $[\![\mu_{s'}(x),\, \mu_{s'}(x^{end})]\!] \in AL^{s'}$ for $x, x^{end} \in \mathcal{V}_{sym}$, then $[\![x,\, x^{end}]\!]$ is added to $AL^{\overline{s}}$. To extend this heuristic to lists, we have to regard several memory entries together. If there is an $\mathtt{ad} \in \mathcal{V}_{\mathcal{P}}$ such that $\mu_s(x_{\mathtt{ad}}) = v_1^{start}$ and $\mu_{s'}(x_{\mathtt{ad}}) = w_1^{start}$ both point to lists of type `ty` but of different lengths $\ell_s \neq \ell_{s'}$ with $\ell_s, \ell_{s'} \geq 1$, then we create a list invariant.

For a state s we say that v_1^{start} *points to a list of type* `ty` *with* n *fields and length* ℓ_s *with allocations* $[\![v_k^{start},\, v_k^{end}]\!]$ *and values* $v_{k,i}$ (for $1 \leq k \leq \ell_s$ and $1 \leq i \leq n$) if the following conditions (a)–(d) hold:

(a) `ty` is an LLVM struct type with subtypes $\mathtt{ty}_i$ and field offsets $\mathit{off}_i \in \mathbb{N}$ for all $1 \leq i \leq n$ such that there exists exactly one $1 \leq j \leq n$ with $\mathtt{ty}_j = \mathtt{ty}{*}$.

(b) There exist pairwise different $[\![v_k^{start},\, v_k^{end}]\!] \in AL^s$ for all $1 \leq k \leq \ell_s$ and $\models \langle s \rangle \Rightarrow v_k^{end} = v_k^{start} + \mathit{size}(\mathtt{ty}) - 1$.

(c) For all $1 \leq k \leq \ell_s$ and $1 \leq i \leq n$ there exist $v_{k,i}^{start}, v_{k,i} \in \mathcal{V}_{sym}$ with $\models \langle s \rangle \Rightarrow v_{k,i}^{start} = v_k^{start} + \mathit{off}_i$ and $(v_{k,i}^{start} \hookrightarrow_{\mathtt{ty}_i} v_{k,i}) \in PT^s$.

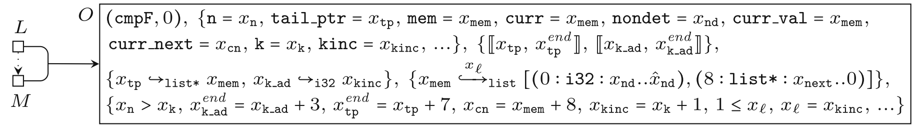
(d) For all $1 \leq k < \ell_s$ we have $\models \langle s \rangle \Rightarrow v_{k,j} = v_{k+1}^{start}$.

Condition (a) states that `ty` is a list type with n fields, where the pointer to the next element is in the j-th field. In (b) we ensure that each list element has a unique allocation of the correct size where v_1^{start} is the start address of the first allocation. Condition (c) requires that for the k-th element, the initial address plus the i-th offset points to a value $v_{k,i}$ of type $\mathtt{ty}_i$. Finally, (d) states that the recursive field of each element indeed points to the initial address of the next element.

Then, for fresh $x_\ell, x_i, \hat{x}_i \in \mathcal{V}_{sym}$, we add the following list invariant to $LI^{\overline{s}}$.

$$x_{\mathtt{ad}} \overset{x_\ell}{\hookrightarrow}_{\mathtt{ty}} [(\mathit{off}_i : \mathtt{ty}_i : x_i..\hat{x}_i)]_{i=1}^{n} \tag{3}$$

To ensure that the allocations expressed by the list invariant are disjoint from all allocations in $AL^{\overline{s}}$, we do not use the list allocations $[\![v_k^{start},\, v_k^{end}]\!]$ to infer generalized allocations in $AL^{\overline{s}}$. Similarly, to create $PT^{\overline{s}}$, we only use entries



$(\texttt{cmpF}, 0)$, $\{\texttt{n} = x_{\texttt{n}}, \texttt{tail\_ptr} = x_{\texttt{tp}}, \texttt{mem} = x_{\texttt{mem}}, \texttt{curr} = x_{\texttt{mem}}, \texttt{nondet} = x_{\texttt{nd}}, \texttt{curr\_val} = x_{\texttt{mem}}, \texttt{curr\_next} = x_{\texttt{cn}}, \texttt{k} = x_{\texttt{k}}, \texttt{kinc} = x_{\texttt{kinc}}, \ldots\}$, $\{[\![x_{\texttt{tp}}, x_{\texttt{tp}}^{end}]\!], [\![x_{\texttt{k\_ad}}, x_{\texttt{k\_ad}}^{end}]\!]\}$,
$\{x_{\texttt{tp}} \hookrightarrow_{\texttt{list*}} x_{\texttt{mem}}, x_{\texttt{k\_ad}} \hookrightarrow_{\texttt{i32}} x_{\texttt{kinc}}\}$, $\{x_{\texttt{mem}} \overset{x_\ell}{\hookrightarrow}_{\texttt{list}} [(0 : \texttt{i32} : x_{\texttt{nd}}..\hat{x}_{\texttt{nd}}), (8 : \texttt{list*} : x_{\texttt{next}}..0)]\}$,
$\{x_{\texttt{n}} > x_{\texttt{k}}, x_{\texttt{k\_ad}}^{end} = x_{\texttt{k\_ad}} + 3, x_{\texttt{tp}}^{end} = x_{\texttt{tp}} + 7, x_{\texttt{cn}} = x_{\texttt{mem}} + 8, x_{\texttt{kinc}} = x_{\texttt{k}} + 1, 1 \le x_\ell, x_\ell = x_{\texttt{kinc}}, \ldots\}$

Fig. 3. Merging of States

$v \hookrightarrow_{\texttt{ty}} w$ from PT^s and $PT^{s'}$ where v is disjoint from the list addresses, i.e., where $\models \langle s \rangle \Rightarrow v < v_k^{start} \vee v > v_k^{end}$ holds for all $1 \le k \le \ell_s$ and analogously for s'. Moreover, we add formulas to $KB^{\overline{s}}$ stating that (A) the length x_ℓ of the list is at least the smaller length of the merged lists, (B) x_ℓ is equal to all variables x which result from merging variables v and w that are equal to the lengths ℓ_s and $\ell_{s'}$ in s and s', and (C) the symbolic variable x_i for the value of the i-th field of the first list element is equal to all variables x with $\mu_s(x) = v_{1,i}$ and $\mu_{s'}(x) = w_{1,i}$ where $v_{1,i}$ and $w_{1,i}$ are the values of the i-th field of the first list element in s and s' (and analogously for the values $\hat{x}_i$ of the last list element):

(A) $\min(\ell_s, \ell_{s'}) \le x_\ell$
(B) $\bigwedge_{x \in \mu_s^{-1}(v) \cap \mu_{s'}^{-1}(w)} x_\ell = x$ for all $v, w \in \mathcal{V}_{sym}$ with $\models \langle s \rangle \Rightarrow v = \ell_s$ and $\models \langle s' \rangle \Rightarrow w = \ell_{s'}$
(C) $\bigwedge_{x \in \mu_s^{-1}(v_{1,i}) \cap \mu_{s'}^{-1}(w_{1,i})} x_i = x$ and $\bigwedge_{x \in \mu_s^{-1}(v_{\ell_s,i}) \cap \mu_{s'}^{-1}(w_{\ell_{s'},i})} \hat{x}_i = x$ for all $1 \le i \le n$

To identify the variables in the list invariant (3) of $\overline{s}$ with the corresponding values in s and s', the instantiations μ_s and $\mu_{s'}$ are extended such that $\mu_s(x_\ell) = \ell_s$, $\mu_{s'}(x_\ell) = \ell_{s'}$, $\mu_s(x_i) = v_{1,i}$, $\mu_{s'}(x_i) = w_{1,i}$, $\mu_s(\hat{x}_i) = v_{\ell_s,i}$, and $\mu_{s'}(\hat{x}_i) = w_{\ell_{s'},i}$ for all $1 \le i \le n$. Similarly, if there already exist list invariants in s and s', for each pair of corresponding variables a new variable is introduced and mapped to its origin by μ_s and $\mu_{s'}$. This adaption of the merging heuristic only concerns the result of merging but not the rules *when* to merge two states. Thus, the same reasoning as in [25] can be used to prove soundness and termination of merging.

In our example, L and M contain lists of length $\ell_L = 1$ and $\ell_M = 2$. To ease the presentation, we re-use variables that are known to be equal instead of introducing fresh variables. If $x_{\texttt{mem}}$ is the variable for the program variable `curr`, we have $\mu_L(x_{\texttt{mem}}) = v_{\texttt{mem}}$ and $\mu_M(x_{\texttt{mem}}) = w_{\texttt{mem}}$. Indeed, $v_{\texttt{mem}}$ resp. $w_{\texttt{mem}}$ points to a list with values $v_{k,i}$ resp. $w_{k,i}$ as defined in (a)–(d): For the type `list` with $n = 2$, $\texttt{ty}_1$=`i32`, $\texttt{ty}_2$=`list*`, off_1=0, off_2=8, and j=2 (see (a)), we have $[\![v_{\texttt{mem}}, v_{\texttt{mem}}^{end}]\!] \in AL^L$ and $[\![v_{\texttt{mem}}, v_{\texttt{mem}}^{end}]\!]$, $[\![w_{\texttt{mem}}, w_{\texttt{mem}}^{end}]\!] \in AL^M$, all consisting of $\mathit{size}(\texttt{list}) = 16$ bytes, see (b). We have $(v_{\texttt{mem}} \hookrightarrow_{\texttt{i32}} v_{\texttt{nd}}), (v_{\texttt{cn}} \hookrightarrow_{\texttt{list*}} 0) \in PT^L$ with $(v_{\texttt{cn}} = v_{\texttt{mem}} + 8) \in KB^L$ and $(v_{\texttt{mem}} \hookrightarrow_{\texttt{i32}} v_{\texttt{nd}}), (v_{\texttt{cn}} \hookrightarrow_{\texttt{list*}} 0), (w_{\texttt{mem}} \hookrightarrow_{\texttt{i32}} w_{\texttt{nd}}), (w_{\texttt{cn}} \hookrightarrow_{\texttt{list*}} v_{\texttt{mem}}) \in PT^M$ with $(v_{\texttt{cn}} = v_{\texttt{mem}} + 8), (w_{\texttt{cn}} = w_{\texttt{mem}} + 8) \in KB^M$ (see (c)), so the first list element in M points to the second one (see (d)). Therefore, when merging L and M to a new state O (see Fig. 3), the lists are merged to a list invariant of variable length x_ℓ and we add the formulas (A) $1 \le x_\ell$ and (B) $x_\ell = x_{\texttt{kinc}}$ to KB^O. By (C), the `i32` value of the first element is identified with $x_{\texttt{nd}}$, since $\mu_L(x_{\texttt{nd}})$ is equal to the first value of the first list element in L and $\mu_M(x_{\texttt{nd}})$ is equal to the first value of the first list element in M. Similarly, the values of the last list elements are identified with 0, as in L and M.

After merging s and s' to a generalized state $\overline{s}$, we continue symbolic execution from $\overline{s}$. The next time we reach the same program position, we might have to merge the corresponding states again. As described in [25], we use a heuristic for constructing the SEG which ensures that after a finite number of iterations, a state is reached that only represents concrete states that are also represented by an *already existing* (more general) state in the SEG. Then symbolic execution can continue from this more general state instead. So with this heuristic, the construction always ends in a complete SEG or an SEG containing the state *ERR*.

We formalized the concept of "generalization" by a symbolic execution rule in [25]. Here, the state $\overline{s}$ is a generalization of s if the conditions $(g1) - (g6)$ hold.

Condition $(g1)$ prevents cycles consisting only of refinement and generalization edges in the graph. Condition $(g2)$ states that the instantiation $\mu \colon \mathcal{V}_{sym}(\overline{s}) \to \mathcal{V}_{sym}(s) \cup \mathbb{Z}$ maps symbolic variables from the more general state $\overline{s}$ to their counterparts from the more specific state s such that they correspond to the same program variable. Conditions $(g3)$–$(g6)$ ensure that all knowledge present in $\overline{KB}$, $\overline{AL}$, $\overline{PT}$, and $\overline{LI}$ still holds in s with the applied instantiation.

generalization with instantiation μ

$$\frac{s = (p,\ LV,\ AL,\ PT,\ LI,\ KB)}{\overline{s} = (p,\ \overline{LV},\ \overline{AL},\ \overline{PT},\ \overline{LI},\ \overline{KB})} \quad \text{if}$$

$(g1)$ s has an incoming evaluation edge

$(g2)$ $\mathit{domain}(LV) = \mathit{domain}(\overline{LV})$ and $LV(\mathtt{var}) = \mu(\overline{LV}(\mathtt{var}))$ for all $\mathtt{var} \in \mathcal{V}_{\mathcal{P}}$ where LV and $\overline{LV}$ are defined

$(g3)$ $\models \langle s \rangle \Rightarrow \mu(\overline{KB})$

$(g4)$ if $[\![x_1,\ x_2]\!] \in \overline{AL}$, then $[\![v_1,\ v_2]\!] \in AL$ with $\models \langle s \rangle \Rightarrow v_1 = \mu(x_1) \wedge v_2 = \mu(x_2)$

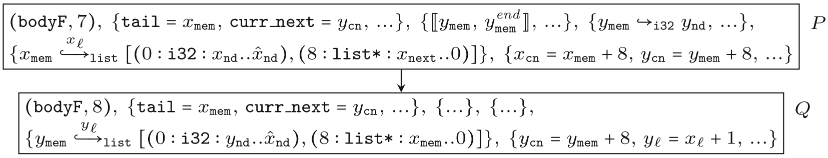
$(g5)$ if $(x_1 \hookrightarrow_{\mathtt{ty}} x_2) \in \overline{PT}$,
then $(v_1 \hookrightarrow_{\mathtt{ty}} v_2) \in PT$ with $\models \langle s \rangle \Rightarrow v_1 = \mu(x_1) \wedge v_2 = \mu(x_2)$

$(g6)$ if $(x_{\mathtt{ad}} \xhookrightarrow{x_\ell}_{\mathtt{ty}} [(\mathit{off}_i : \mathtt{ty}_i : x_i..\hat{x}_i)]_{i=1}^{n}) \in \overline{LI}$,
then either $(v_{\mathtt{ad}} \xhookrightarrow{v_\ell}_{\mathtt{ty}} [(\mathit{off}_i : \mathtt{ty}_i : v_i..\hat{v}_i)]_{i=1}^{n}) \in LI$ with
- $\models \langle s \rangle \Rightarrow v_{\mathtt{ad}} = \mu(x_{\mathtt{ad}}) \wedge v_\ell = \mu(x_\ell)$ and
- $\models \langle s \rangle \Rightarrow v_i = \mu(x_i) \wedge \hat{v}_i = \mu(\hat{x}_i)$ for all $1 \leq i \leq n$,

or v_1^{start} points to a list of type $\mathtt{ty}$ and length ℓ with allocations $[\![v_k^{start},\ v_k^{end}]\!]$ and values $v_{k,i}$ (for $1 \leq k \leq \ell, 1 \leq i \leq n$) such that
- $\models \langle s \rangle \Rightarrow v_1^{start} = \mu(x_{\mathtt{ad}}) \wedge \ell = \mu(x_\ell)$,
- $\models \langle s \rangle \Rightarrow v_{1,i} = \mu(x_i) \wedge v_{\ell,i} = \mu(\hat{x}_i)$ for all $1 \leq i \leq n$, and
- if $(z_1 \hookrightarrow_{\mathtt{ty}} z_2) \in \overline{PT}$,
then $\models \langle s \rangle \Rightarrow \mu(z_1) < v_k^{start} \vee \mu(z_1) > v_k^{end}$ for all $1 \leq k \leq \ell$.

Condition $(g6)$ is new compared to [25] and takes list invariants into account. So for every list invariant $\overline{l}$ of $\overline{s}$ there is either a corresponding list invariant l in s such that lists represented by l in s are also represented by $\overline{l}$ in $\overline{s}$, or there is a concrete list in s that is represented by $\overline{l}$ in $\overline{s}$. The last condition of the latter case ensures that disjointness between the memory domains of $\overline{PT}$ and $\overline{LI}$ is preserved. See [18] for the soundness proof of the extended generalization rule, i.e., that every concrete state represented by s is also represented by $\overline{s}$.

Our merging technique always yields generalizations according to this rule, i.e., the edges from L and M to O in Fig. 3 are generalization edges. Here, one



$(\texttt{bodyF}, 7),\ \{\texttt{tail} = x_{\texttt{mem}},\ \texttt{curr\_next} = y_{\texttt{cn}},\ \ldots\},\ \{[\![y_{\texttt{mem}},\ y_{\texttt{mem}}^{end}]\!],\ \ldots\},\ \{y_{\texttt{mem}} \hookrightarrow_{\texttt{i32}} y_{\texttt{nd}},\ \ldots\},$
$\{x_{\texttt{mem}} \overset{x_\ell}{\hookrightarrow}_{\texttt{list}} [(0 : \texttt{i32} : x_{\texttt{nd}}..\hat{x}_{\texttt{nd}}), (8 : \texttt{list*} : x_{\texttt{next}}..0)]\},\ \{x_{\texttt{cn}} = x_{\texttt{mem}} + 8,\ y_{\texttt{cn}} = y_{\texttt{mem}} + 8,\ \ldots\}$

$(\texttt{bodyF}, 8),\ \{\texttt{tail} = x_{\texttt{mem}},\ \texttt{curr\_next} = y_{\texttt{cn}},\ \ldots\},\ \{\ldots\},\ \{\ldots\},$
$\{y_{\texttt{mem}} \overset{y_\ell}{\hookrightarrow}_{\texttt{list}} [(0 : \texttt{i32} : y_{\texttt{nd}}..\hat{x}_{\texttt{nd}}), (8 : \texttt{list*} : x_{\texttt{mem}}..0)]\},\ \{y_{\texttt{cn}} = y_{\texttt{mem}} + 8,\ y_\ell = x_\ell + 1,\ \ldots\}$

Fig. 4. Extending a List Invariant

chooses μ_L and μ_M such that $\mu_L(x_{\texttt{mem}}) = v_{\texttt{mem}}$, $\mu_L(x_\ell) = 1$, $\mu_L(x_{\texttt{nd}}) = v_{\texttt{nd}}$, $\mu_L(\hat{x}_{\texttt{nd}}) = v_{\texttt{nd}}$, $\mu_L(x_{\texttt{next}}) = 0$, $\mu_M(x_{\texttt{mem}}) = w_{\texttt{mem}}$, $\mu_M(x_\ell) = 2$, $\mu_M(x_{\texttt{nd}}) = w_{\texttt{nd}}$, $\mu_L(\hat{x}_{\texttt{nd}}) = v_{\texttt{nd}}$, and $\mu_M(x_{\texttt{next}}) = v_{\texttt{mem}}$. In both cases, all conditions of the second case of ($g6$) with $\ell_L = 1$ and $\ell_M = 2$ are satisfied. With $\mu_L(x_{\texttt{kinc}}) = 1$ resp. $\mu_M(x_{\texttt{kinc}}) = 2$, we also have $\models \langle L \rangle \Rightarrow \mu_L(x_\ell) = \mu_L(x_{\texttt{kinc}})$ resp. $\models \langle M \rangle \Rightarrow \mu_M(x_\ell) = \mu_M(x_{\texttt{kinc}})$.

3.2 Adapting List Invariants

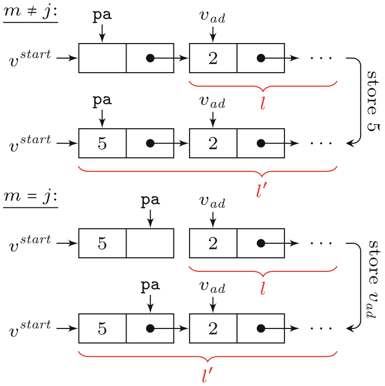
To handle and modify list invariants, three of our symbolic execution rules have to be changed. Section 3.2.1 presents a variant of the `store` rule where the list invariant is *extended* by an element. In Sect. 3.2.2, we adapt the `load` rule to load values from the first list element and we present a variant of the `getelementptr` rule for list *traversal*. Soundness of our new rules is proved in [18]. For all other instructions, the symbolic execution rules from [25] remain unchanged.

3.2.1 List Extension

After merging L and M, symbolic execution continues from the more general state O in Fig. 3. Here, the values of `k` and `kinc` and the length of the list are not concrete but any positive (resp. non-negative) value with $x_\ell = x_{\texttt{kinc}} = x_{\texttt{k}} + 1$. The symbolic execution of O is similar to the steps from B to J in Sect. 3 (see Fig. 1). First, the value $x_{\texttt{kinc}}$ stored at `k_ad` is loaded to `k`. To distinguish whether `k < n` still holds, the next state is refined. From the refined state with `k < n`, we enter the loop body again. A new block $[\![y_{\texttt{mem}},\ y_{\texttt{mem}}^{end}]\!]$ of 16 bytes is allocated and $y_{\texttt{mem}}$ is assigned to `mem` and `curr`. Then, a new unknown value $y_{\texttt{nd}}$ is assigned to `nondet`. The address of the `i32` value of the current element (equal to $y_{\texttt{mem}}$) is computed by the first `getelementptr` instruction of the loop and the value $y_{\texttt{nd}}$ of `nondet` is stored at it. The second `getelementptr` instruction computes the address $y_{\texttt{cn}}$ of the recursive field and results in State P in Fig. 4, where $y_{\texttt{cn}} = y_{\texttt{mem}} + 8$ is added to KB^P. Now, `store` sets the address of the `next` field to the head of the list created in the previous iteration. Since this instruction extends the list by an element, instead of adding $y_{\texttt{cn}} \hookrightarrow_{\texttt{list*}} x_{\texttt{mem}}$ to PT^Q, we extend the list invariant: The length is set to y_ℓ and identified with $x_\ell + 1$ in KB^Q. The pointer $x_{\texttt{mem}}$ to the first element is replaced by $y_{\texttt{mem}}$, while the first recursive field in the list gets the value $x_{\texttt{mem}}$. Since $(y_{\texttt{mem}} \hookrightarrow_{\texttt{i32}} y_{\texttt{nd}}) \in PT^P$, $y_{\texttt{nd}}$ is the value of the first `i32` integer in the list. We remove all entries from PT^Q that are already contained in the new list invariant, e.g., $y_{\texttt{mem}} \hookrightarrow_{\texttt{i32}} y_{\texttt{nd}}$.

To formalize this adaption of list invariants, we introduce a modified rule for `store` in addition to the one in [25]. It handles the case where there is a concrete list at some address v^{start}, `pa` points to the m-th field of this list's first element, one wants to store a value t at the address `pa`, and one already has a list invariant l for the "tail" of the list in the j-th field (if $m \neq j$) resp. for the list at the address t (if $m = j$). In all other cases, the ordinary `store` rule is applied.

More precisely, let the list invariant l describe a list of length v_l at the address v_{ad}. Then l is replaced by a new list invariant l' which describes the list at the address v^{start} after storing t at the address `pa`. Irrespective of whether $m \neq j$ or $m = j$, the resulting list at v^{start} has the list at v_{ad} as its "tail" and thus, its length v'_ℓ is $v_\ell + 1$. We prevent sharing of different elements by removing the allocation $[\![v^{start},\ v^{end}]\!]$ of the list and all points-to information of pointers in $[\![v^{start},\ v^{end}]\!]$.



list extension (p : **"**`store ty` t**,** `ty* pa`**"**, $t \in \mathcal{V}_\mathcal{P} \cup \mathbb{N}$, `pa` $\in \mathcal{V}_\mathcal{P}$)

$$\frac{s = (p,\ LV,\ AL,\ PT,\ LI,\ KB)}{s' = (p^+,\ LV,\ AL\backslash\{[\![v^{start},\ v^{end}]\!]\},\ PT',\ LI\backslash\{l\} \cup \{l'\},\ KB')} \quad \text{if}$$

- there is $l = (v_{ad} \xhookrightarrow{v_\ell}_{\texttt{lty}} [(\mathit{off}_i : \texttt{lty}_i : w_i..\hat{w}_i)]_{i=1}^{n}) \in LI$ with $\texttt{lty}_j = \texttt{lty*}$
- there is $[\![v^{start},\ v^{end}]\!] \in AL$ with $\models \langle s \rangle \Rightarrow v^{end} = v^{start} + \mathit{size}(\texttt{lty}) - 1$
- there exists $1 \leq m \leq n$ such that $\texttt{ty} = \texttt{lty}_m$ and $\models \langle s \rangle \Rightarrow LV(\texttt{pa}) = v^{start} + \mathit{off}_m$
- $\models \langle s \rangle \Rightarrow v_{ad} = v_j$ if $m \neq j$ and $\models \langle s \rangle \Rightarrow v_{ad} = LV(t)$ if $m = j$
- for all $1 \leq i \leq n$ with $i \neq m$ there exist $v_i^{start}, v_i \in \mathcal{V}_{sym}$ with $\models \langle s \rangle \Rightarrow v_i^{start} = v^{start} + \mathit{off}_i$ and $(v_i^{start} \hookrightarrow_{\texttt{lty}_i} v_i) \in PT$
- $PT' = \{(x_1 \hookrightarrow_{\texttt{sy}} x_2) \in PT \mid \models \langle s \rangle \Rightarrow (v^{end} < x_1) \vee (x_1 + \mathit{size}(\texttt{sy}) - 1 < v^{start})\}$
- $l' = (v^{start} \xhookrightarrow{v'_\ell}_{\texttt{lty}} [(\mathit{off}_i : \texttt{lty}_i : v_i..\hat{w}_i)]_{i=1}^{n})$
- $KB' = KB \cup \{v_m = LV(t),\ v'_\ell = v_\ell + 1\}$, where v_m, v'_ℓ are fresh

3.2.2 List Traversal

After the current element $y_{\texttt{mem}}$ is stored at $x_{\texttt{tp}}$ and the value $x_{\texttt{kinc}}$ of `k` is incremented to $y_{\texttt{kinc}}$ and stored at $x_{\texttt{k\_ad}}$, we reach a state R at position $(\texttt{cmpF}, 0)$ by the branch instruction. However, our already existing state O is more general than R, i.e., we can draw a generalization edge from R to O using the generalization rule with the instantiation μ_R where $\mu_R(x_{\texttt{mem}}) = y_{\texttt{mem}}$, $\mu_R(x_{\texttt{nd}}) = y_{\texttt{nd}}$, $\mu_R(x_{\texttt{cn}}) = y_{\texttt{cn}}$, $\mu_R(x_{\texttt{k}}) = x_{\texttt{kinc}}$, $\mu_R(x_{\texttt{kinc}}) = y_{\texttt{kinc}}$, $\mu_R(x_\ell) = y_\ell$, $\mu_R(\hat{x}_{\texttt{nd}}) = \hat{x}_{\texttt{nd}}$, and $\mu_R(x_{\texttt{next}}) = x_{\texttt{mem}}$. Thus, the cycle of the first loop closes here.

$$\boxed{\begin{array}{l}(\mathtt{bodyW}, 1),\ \{\mathtt{ptr} = x_{\mathtt{ptr}},\ \mathtt{curr'} = x_{\mathtt{next}},\ \mathtt{next\_ptr} = x_{\mathtt{np}},\ \mathtt{next} = x_{\mathtt{next}},\ \ldots\},\\ \{\llbracket x_{\mathtt{ptr}},\ x_{\mathtt{ptr}}^{end}\rrbracket,\ \ldots\},\ \{x_{\mathtt{ptr}} \hookrightarrow_{\mathtt{list*}} x_{\mathtt{next}},\ \ldots\},\\ \{x_{\mathtt{mem}} \overset{x_\ell}{\hookrightarrow}_{\mathtt{list}} [(0 : \mathtt{i32} : x_{\mathtt{nd}}..\hat{x}_{\mathtt{nd}}), (8 : \mathtt{list*} : x_{\mathtt{next}}..0)]\},\\ \{x_{\mathtt{np}} = x_{\mathtt{mem}} + 8,\ x_{\mathtt{next}} \geq 1,\ \ldots\}\end{array}}\ U$$

$$\downarrow$$

$$\boxed{\begin{array}{l}(\mathtt{bodyW}, 2),\ \{\mathtt{ptr} = x_{\mathtt{ptr}},\ \mathtt{curr'} = x_{\mathtt{next}},\ \mathtt{next\_ptr} = x'_{\mathtt{np}},\ \mathtt{next} = x_{\mathtt{next}},\ \ldots\},\\ \{\llbracket x_{\mathtt{ptr}},\ x_{\mathtt{ptr}}^{end}\rrbracket,\ \llbracket x_{\mathtt{mem}},\ x_{\mathtt{mem}}^{end}\rrbracket,\ \ldots\},\ \{x_{\mathtt{ptr}} \hookrightarrow_{\mathtt{list*}} x_{\mathtt{next}},\ x_{\mathtt{mem}} \hookrightarrow_{\mathtt{i32}} x_{\mathtt{nd}},\ x_{\mathtt{np}} \hookrightarrow_{\mathtt{list*}} x_{\mathtt{next}},\ \ldots\},\\ \{x_{\mathtt{next}} \overset{x'_\ell}{\hookrightarrow}_{\mathtt{list}} [(0 : \mathtt{i32} : x'_{\mathtt{nd}}..\hat{x}_{\mathtt{nd}}), (8 : \mathtt{list*} : x'_{\mathtt{next}}..0)]\},\\ \{x_{\mathtt{np}} = x_{\mathtt{mem}} + 8,\ x_{\mathtt{mem}}^{end} = x_{\mathtt{mem}} + 15,\ x'_{\mathtt{np}} = x_{\mathtt{next}} + 8,\ x'_\ell = x_\ell - 1,\ \ldots\}\end{array}}\ V$$

Fig. 5. Traversing a List Invariant

$$\boxed{\begin{array}{l}(\mathtt{initPtr}, 0),\ \{\mathtt{tail\_ptr} = x_{\mathtt{tp}},\ \ldots\},\ \{\ldots\},\ \{x_{\mathtt{tp}} \hookrightarrow_{\mathtt{list*}} x_{\mathtt{mem}},\ \ldots\},\\ \{x_{\mathtt{mem}} \overset{x_\ell}{\hookrightarrow}_{\mathtt{list}} [(0 : \mathtt{i32} : x_{\mathtt{nd}}..\hat{x}_{\mathtt{nd}}), (8 : \mathtt{list*} : x_{\mathtt{next}}..0)]\},\ \{\ldots\}\end{array}}\ S$$

$$\vdots$$

$$\boxed{\begin{array}{l}(\mathtt{cmpW}, 0),\ \{\mathtt{ptr} = x_{\mathtt{ptr}},\ \mathtt{curr'} = x_{\mathtt{mem}},\ \mathtt{next\_ptr} = x_{\mathtt{np}},\\ \mathtt{next} = x_{\mathtt{next}},\ \ldots\},\ \{\llbracket x_{\mathtt{ptr}},\ x_{\mathtt{ptr}}^{end}\rrbracket,\ \ldots\},\ \{x_{\mathtt{ptr}} \hookrightarrow_{\mathtt{list*}} x_{\mathtt{next}},\ \ldots\},\\ \{x_{\mathtt{mem}} \overset{x_\ell}{\hookrightarrow}_{\mathtt{list}} [(0 : \mathtt{i32} : x_{\mathtt{nd}}..\hat{x}_{\mathtt{nd}}), (8 : \mathtt{list*} : x_{\mathtt{next}}..0)]\},\\ \{x_{\mathtt{np}} = x_{\mathtt{mem}} + 8,\ \ldots\}\end{array}}\ T$$

```
initPtr:
  0: tail' = load list*, list** tail_ptr
  1: store list* tail', list** ptr
  2: br label cmpW

cmpW:
  0: str = load list*, list** ptr
  1: notnull = icmp ne list* str, null
  2: br i1 notnull, label bodyW, label ret

bodyW:
  0: curr' = bitcast list* str to i8*
  1: next_ptr = getelementptr i8, i8* curr', i64 8
  2: next_ptr' = bitcast i8* next_ptr to list**
  3: next = load list*, list** next_ptr'
  4: store list* next, list** ptr
  5: br label cmpW
```

As mentioned, in the path from O to R there is a state at position $(\mathtt{cmpF}, 1)$ which is refined (similar to State C). If `k < n` holds, we reach R. The other path with `k` $\not<$`n` leads out of the `for` loop to the block `initPtr` followed by the `while` loop (see State S and the corresponding LLVM code on the side). The value $x_{\mathtt{mem}}$ at address `tail_ptr` is loaded to `tail'` and stored at a new pointer variable `ptr`. State T is reached after the first iteration of the `while` loop body. Here, block `cmpW` loads the value $x_{\mathtt{mem}}$ stored at `ptr` to `str`. Since it is not the null pointer, we enter `bodyW`, which corresponds to the body of the `while` loop. First, $x_{\mathtt{mem}}$ is cast to an `i8` pointer. Then `getelementptr` computes a pointer $x_{\mathtt{np}}$ to the next element by adding 8 bytes to $x_{\mathtt{mem}}$. After another cast back to a `list*` pointer, we load the content of the new pointer to `next`. To this end, we need the following new variant of the `load` rule to load values that are described by a list invariant.

`load` from list invariant (p : "`x = load ty, ty* ad_i`", $\mathtt{x}, \mathtt{ad\_i} \in \mathcal{V}_{\mathcal{P}}$)

$$\frac{s = (p,\ LV,\ AL,\ PT,\ LI,\ KB)}{s' = (p^+,\ LV[\mathtt{x} := w],\ AL,\ PT,\ LI,\ KB \cup \{w = v_i\})} \quad \text{if } w \in \mathcal{V}_{sym} \text{ is fresh and}$$

- there is $l = (v_{ad} \overset{v_\ell}{\hookrightarrow}_{\mathtt{ty}} [(\mathit{off}_i : \mathtt{ty}_i : v_i..\hat{v}_i)]_{i=1}^{n}) \in LI$
- there exists $1 \leq i \leq n$ such that $\mathtt{ty} = \mathtt{ty}_i$ and $\models \langle s \rangle \Rightarrow LV(\mathtt{ad\_i}) = v_{ad} + \mathit{off}_i$

With this new `load` rule, the content of the new pointer is identified as $x_{\mathtt{next}}$. It is loaded to `next` and stored at $x_{\mathtt{ptr}}$. Then we return to the block `cmpW` (State T). Merging T with its predecessor at the same program position is not possible yet since the domains of the respective LV functions do not coincide. Now, $x_{\mathtt{next}}$ is loaded to `str` and compared to the null pointer. Since we do not have information about $x_{\mathtt{next}}$, T's successor state is refined to a state with $x_{\mathtt{next}} = 0$ (which starts a path out of the loop to a return state), and to a state with $x_{\mathtt{next}} \geq 1$, which reaches U after a few evaluation steps, see Fig. 5. Now, `getelementptr` computes the pointer $x'_{\mathtt{np}} = x_{\mathtt{next}} + 8$ to the third element of the list, which is assigned to `next_ptr`. $\langle U \rangle$ contains $x_\ell \geq 2$ since the first and the last pointer value are known to be different ($x_{\mathtt{next}} \neq 0$). This information is crucial for creating a new list invariant starting at $x_{\mathtt{next}}$, which is used in the next iteration of the loop. Therefore, if our list invariant did not contain variables for the first and the last pointer, we could not prove termination of the program. In such a case where the pointer to the third element of a list invariant is computed and the length of the list is at least two, we *traverse* the list invariant to retain the correspondence between the computed pointer $x'_{\mathtt{np}}$ and the new list invariant. In the resulting state V, we represent the first list element by an allocation $[\![x_{\mathtt{mem}}, x_{\mathtt{mem}}^{end}]\!]$ and preserve all knowledge about this element that was encoded in the list invariant ($x_{\mathtt{mem}}^{end} = x_{\mathtt{mem}} + 15$, $x_{\mathtt{mem}} \hookrightarrow_{\mathtt{i32}} x_{\mathtt{nd}}$, $x_{\mathtt{np}} \hookrightarrow_{\mathtt{list*}} x_{\mathtt{next}}$). Moreover, we adapt the list invariant such that it now represents the list at $x_{\mathtt{next}}$ (i.e., without its first element) starting with the value $x'_{\mathtt{nd}}$. We also relate the length of the new list invariant to the length of the former one ($x'_\ell = x_\ell - 1$).

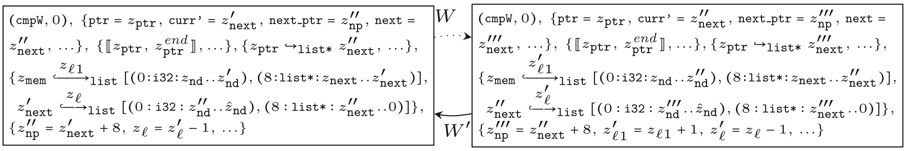
Thus, in addition to the rule for `getelementptr` in [25], we now introduce rules for list traversal via `getelementptr`. The rule below handles the case where the address calculation is based on the type `i8` and the `getelementptr` instruction adds the number of bytes given by the term t to the address `pa`. Here, the offsets in our list invariants are needed to compute the address of the accessed field. We also have similar rules for list traversal via field access (i.e., where the next element is accessed using `curr'->next` as in the `for` loop) and for the case where we cannot prove that the length v_ℓ of the list is at least 2, see [18].

list traversal (p : "`pb = getelementptr i8, i8* pa, i`m t", $t \in \mathcal{V}_\mathcal{P} \cup \mathbb{N}$, $\texttt{pa}, \texttt{pb} \in \mathcal{V}_\mathcal{P}$)

$$\frac{s = (p,\ LV,\ AL,\ PT,\ LI,\ KB)}{s' = (p^+,\ LV[\texttt{pb} := w_j^{start}],\ AL \cup [\![v^{start},\ v^{end}]\!],\ PT',\ LI \backslash \{l\} \cup \{l'\},\ KB')} \quad \text{if}$$

- there is $l = (v_{ad} \xhookrightarrow{v_\ell}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : v_i..\hat{v}_i)]_{i=1}^n) \in LI$ with $\texttt{ty}_j = \texttt{ty*}$, $\models \langle s \rangle \Rightarrow LV(\texttt{pa}) = v_j$, $\models \langle s \rangle \Rightarrow LV(t) = \mathit{off}_j$, and $\models \langle s \rangle \Rightarrow v_\ell \geq 2$
- $PT' = PT \cup \{(v_i^{start} \hookrightarrow_{\texttt{ty}_i} v_i) \mid 1 \leq i \leq n\}$
- $l' = (w^{start} \xhookrightarrow{w_\ell}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : w_i..\hat{v}_i)]_{i=1}^n)$
- $KB' = KB \cup \{v^{start} = v_{ad},\ v^{end} = v^{start} + \mathit{size}(\texttt{ty}) - 1,\ w^{start} = v_j,\ w_\ell = v_\ell - 1,\ w_j^{start} = w^{start} + \mathit{off}_j\} \cup \{v_i^{start} = v_{ad} + \mathit{off}_i \mid 1 \leq i \leq n\}$
- $v^{start}, v^{end}, v_1^{start}, \ldots, v_n^{start}, w^{start}, w_\ell, w_j^{start}, w_1, \ldots, w_n \in \mathcal{V}_{sym}$ are fresh

We continue the symbolic execution of State V in our example and finally obtain a complete SEG with a path from a state W at the position $(\texttt{cmpW}, 0)$ to the next state W' at this position, and a generalization edge back from W' to W using an instantiation $\mu_{W'}$. Both W and W' contain a list invariant similar to T where instead of the length x_ℓ in T, we have the symbolic variables z_ℓ and z'_ℓ in W and W', where $\mu_{W'}(z_\ell) = z'_\ell$ (see [18] for more details).


$(\texttt{cmpW}, 0)$, $\{\texttt{ptr} = z_{\texttt{ptr}}, \texttt{curr'} = z'_{\texttt{next}}, \texttt{next\_ptr} = z''_{\texttt{np}}, \texttt{next} = z''_{\texttt{next}}, \ldots\}$, $\{[\![z_{\texttt{ptr}}, z_{\texttt{ptr}}^{end}]\!], \ldots\}$, $\{z_{\texttt{ptr}} \hookrightarrow_{\texttt{list*}} z''_{\texttt{next}}, \ldots\}$,
$\{z_{\texttt{mem}} \xhookrightarrow{z_{\ell 1}}_{\texttt{list}} [(0:\texttt{i32}:z_{\texttt{nd}}..z'_{\texttt{nd}}), (8:\texttt{list*}:z_{\texttt{next}}..z'_{\texttt{next}})]$,
$z'_{\texttt{next}} \xhookrightarrow{z_\ell}_{\texttt{list}} [(0:\texttt{i32}:z''_{\texttt{nd}}..\hat{z}_{\texttt{nd}}), (8:\texttt{list*}:z''_{\texttt{next}}..0)]\}$,
$\{z''_{\texttt{np}} = z'_{\texttt{next}} + 8, z_\ell = z'_\ell - 1, \ldots\}$
W
W'
$(\texttt{cmpW}, 0)$, $\{\texttt{ptr} = z_{\texttt{ptr}}, \texttt{curr'} = z''_{\texttt{next}}, \texttt{next\_ptr} = z'''_{\texttt{np}}, \texttt{next} = z'''_{\texttt{next}}, \ldots\}$, $\{[\![z_{\texttt{ptr}}, z_{\texttt{ptr}}^{end}]\!], \ldots\}$, $\{z_{\texttt{ptr}} \hookrightarrow_{\texttt{list*}} z'''_{\texttt{next}}, \ldots\}$,
$\{z_{\texttt{mem}} \xhookrightarrow{z'_{\ell 1}}_{\texttt{list}} [(0:\texttt{i32}:z_{\texttt{nd}}..z''_{\texttt{nd}}), (8:\texttt{list*}:z_{\texttt{next}}..z''_{\texttt{next}})]$,
$z''_{\texttt{next}} \xhookrightarrow{z'_\ell}_{\texttt{list}} [(0:\texttt{i32}:z'''_{\texttt{nd}}..\hat{z}_{\texttt{nd}}), (8:\texttt{list*}:z'''_{\texttt{next}}..0)]\}$,
$\{z'''_{\texttt{np}} = z''_{\texttt{next}} + 8, z'_{\ell 1} = z_{\ell 1} + 1, z'_\ell = z_\ell - 1, \ldots\}$

4 Proving Termination

To prove termination of a program $\mathcal{P}$, as in [25] the cycles of the SEG are translated to an integer transition system whose termination implies termination of $\mathcal{P}$. The edges of the SEG are transformed into ITS transitions whose application conditions consist of the state formulas $\langle s \rangle$ and equations to identify corresponding symbolic variables of the different states. For evaluation and refinement edges, the symbolic variables do not change. For generalization edges, we use the instantiation μ to identify corresponding symbolic variables. In our example, the ITS has cyclic transitions of the following form:

$$\begin{array}{rcll}
O(x_{\texttt{n}}, x_{\texttt{k}}, x_{\texttt{kinc}}, \ldots) & \to^+ & R(x_{\texttt{n}}, x_{\texttt{k}}, x_{\texttt{kinc}}, \ldots) & \mid \quad x_{\texttt{kinc}} = x_{\texttt{k}} + 1 \wedge x_{\texttt{n}} > x_{\texttt{k}} \wedge \ldots \\
R(x_{\texttt{n}}, x_{\texttt{k}}, x_{\texttt{kinc}}, \ldots) & \to & O(x_{\texttt{n}}, x_{\texttt{kinc}}, \ldots) & \\
W(z_\ell, z'_\ell, \ldots) & \to^+ & W'(z_\ell, z'_\ell, \ldots) & \mid \quad z_\ell = z'_\ell - 1 \wedge z_\ell \geq 1 \wedge \ldots \\
W'(z_\ell, z'_\ell, \ldots) & \to & W(z'_\ell, \ldots) &
\end{array}$$

The first cycle resulting from the generalization edge from R to O terminates since `k` is increased until it reaches `n`. The generalization edge yields a condition identifying $x_{\texttt{kinc}}$ in R with $x_{\texttt{k}}$ in O, since $\mu_R(x_{\texttt{k}}) = x_{\texttt{kinc}}$. With the conditions $x_{\texttt{kinc}} = x_{\texttt{k}} + 1$ and $x_{\texttt{n}} > x_{\texttt{k}}$ (from KB^O), the resulting transitions of the ITS are terminating. The second cycle from the generalization edge from W' to W

terminates since the length of the list starting with `curr'` decreases. Although there is no program variable for the length, due to our list invariants the states contain variables for this length, which are also passed to the ITS. Thus, the ITS contains the variable z_ℓ (where z_ℓ in W is identified with z'_ℓ in W' due to $\mu_{W'}(z_\ell) = z'_\ell$). Since the condition $z'_\ell = z_\ell - 1$ is obtained on the path from W to W' and $z_\ell \geq 1$ is part of $\langle W \rangle$ due to the list invariant with length z_ℓ in LI^W, the resulting transitions of the ITS clearly terminate. Analogous to [25, Cor. 11 and Thm. 13], we obtain the following theorem. To prove that a complete SEG represents all program paths, in [25] we used the LLVM semantics defined by the Vellvm project [26]. One now also has to prove soundness of those symbolic execution rules which were modified due to the new concept of list invariants (i.e., generalization, list extension, and list traversal), see [18].

Theorem 1 (Memory Safety and Termination). *Let $\mathcal{P}$ be a program with a complete SEG $\mathcal{G}$. Since a complete SEG does not contain ERR, $\mathcal{P}$ is memory safe for all concrete states represented by the states in $\mathcal{G}$.*[4] *If the ITS corresponding to $\mathcal{G}$ is terminating, then $\mathcal{P}$ is also terminating for all states represented by $\mathcal{G}$.*

5 Conclusion, Related Work, and Evaluation

We presented a new approach for automated proofs of memory safety and termination of C/LLVM-programs on lists. It first constructs a symbolic execution graph (SEG) which overapproximates all program runs. Afterwards, an integer transition system (ITS) is generated from this graph whose termination is proved using standard techniques. The main idea of our new approach is the extension of the states in the SEG by suitable *list invariants*. We developed techniques to infer and modify list invariants automatically during the symbolic execution.

During the construction of the SEG, the list invariants abstract from a concrete number of memory allocations to a list of allocations of variable length while preserving knowledge about some of the contents (the values of the fields of the first and the last element) and the list shape (the start address of the first element, the list length, and the content of the last recursive pointer which allows us to distinguish between cyclic and acyclic lists). They also contain information on the memory arrangement of the list fields which is needed for programs that access fields via pointer arithmetic. The symbolic variables for the list length and the first and last values of list elements are preserved when generating an ITS from the SEG. Thus, they can be used in the termination proof of the ITS (e.g., the variables for list length can occur in ranking functions).

In [5,6,22] we developed a technique for termination analysis of Java, based on a program transformation to *integer term rewrite systems* instead of ITSs. This approach does not require specific list invariants as recursive data structures on the heap are abstracted to terms. However, these terms are unsuitable for

[4] Our approach can only *prove* but not *disprove* memory safety, i.e., a SEG with the state *ERR* just means that we failed in showing memory safety.

C, since they cannot express memory allocations and the connection to their contents.

Separation logic predicates for termination of list programs were also used in [1], but their list predicates only consider the list length and the recursive field, but no other fields or offsets. The tools Cyclist [24] and HipTNT+ [19] are integrated in separation logic systems which also allow to define heap predicates. However, they require annotations and hints which parameters of the list predicates are needed as a termination measure. The tool 2LS [20] also provides basic support for dynamic data structures. But all these approaches are not suitable if termination depends on the contents or the shape of data structures combined with pointer arithmetic. In [10], programs can be annotated with arithmetic and structural properties to reason about termination. In contrast, our approach does not need hints or annotations, but finds termination arguments fully automatically.

We implemented our approach in AProVE [25]. While C programs with lists are very common, existing tools can hardly prove their termination. Therefore, the current benchmark collections for termination analysis contain almost no list programs. In 2017, a benchmark set[5] of 18 typical C-programs on lists was added to the *Termination* category of the *Competition on Software Verification* (*SV-COMP*) [3], where 9 of them are terminating. Two of these 9 programs do not need list invariants, because they just create a list without operating on it afterwards. The remaining seven terminating programs create a list and then traverse it, search for a value, or append lists and compute the length afterwards. Only few tools in *SV-COMP* produced correct termination proofs for programs from this set: HipTNT+ and 2LS failed for all of them. CPAchecker [2] and PeSCo [23] proved termination and non-termination for one of these programs in 2020. UAutomizer [8] proved termination for two and non-termination for seven programs. The termination proofs of CPAchecker, PeSCo, and UAutomizer only concern the programs that just create a list. Our new version of AProVE is the only termination prover[6] that succeeds if termination depends on the shape or contents of a list after its creation. Note that for non-termination, a proof is a single non-terminating program path, so here list invariants are less helpful.

For the *Termination Competition* [15] 2022, we submitted 18 terminating C programs on lists[7] (different from the ones at *SV-COMP*), where two of them just create a list. Three traverse it afterwards (by a loop or recursion), and ten search for a value, where for nine, also the list contents are relevant for termination. Three programs perform common operations like inserting or deleting an element. UAutomizer proves termination for a program that just creates a list but not for programs operating on the list afterwards. With our approach, AProVE succeeds on 17 of the 18 programs. Overall, AProVE and UAutomizer were the two

[5] https://github.com/sosy-lab/sv-benchmarks/tree/master/c/termination-memory-linkedlists.

[6] We did not compare with the tool VeriFuzz [21], since it does not prove termination but only tests for non-termination and thus, it is unsound for inferring termination.

[7] https://github.com/TermCOMP/TPDB/tree/master/C/Hensel_22.

most powerful tools for termination of C in *SV-COMP* 2022 and the *Termination Competition* 2022, with UAutomizer winning the former and AProVE winning the latter competition. To download AProVE, run it via its web interface, and for details on our experiments, see https://aprove-developers.github.io/recursive_structs.

	SV-C T.	**SV-C Non-T.**	**TermCmp T.**
AProVE	7 (of 9)	5 (of 9)	17 (of 18)
UAutomizer	2 (of 9)	7 (of 9)	1 (of 18)

References

1. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.W.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_35
2. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
3. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13244, pp. 375–402. Springer, Cham (2022). http://sv-comp.sosy-lab.org/2022/. https://doi.org/10.1007/978-3-030-99527-0_20
4. Bozga, M., Iosif, R., Perarnau, S.: Quantitative separation logic and programs with lists. J. Aut. Reason. **45**(2), 131–156 (2010). http://dx.doi.org/10.1007/s10817-010-9179-9
5. Brockschmidt, M., Otto, C., Giesl, J.: Modular termination proofs of recursive Java Bytecode programs by term rewriting. In: Schmidt-Schauss, S. (ed.) RTA 2011, LIPIcs 10, pp. 155–170 (2011). http://dx.doi.org/10.4230/LIPIcs.RTA.2011.155
6. Brockschmidt, M., Musiol, R., Otto, C., Giesl, J.: Automated termination proofs for Java programs with cyclic data. In: Madhusudan, P.., Seshia, Sanjit A.. (eds.) CAV 2012. LNCS, vol. 7358, pp. 105–122. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_13
7. Brockschmidt, M., Cook, B., Ishtiaq, S., Khlaaf, H., Piterman, N.: T2: temporal property verification. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 387–393. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_22
8. Chen, Y.-F., et al.: Advanced automata-based algorithms for program termination checking. In: Foster, J.S., Grossman, D. (eds.) PLDI 2018, pp. 135–150 (2018). http://dx.doi.org/10.1145/3192366.3192405
9. Clang: http://clang.llvm.org
10. David, C., Kroening, D., Lewis, M.: Propositional reasoning about safety and termination of heap-manipulating programs. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 661–684. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_27
11. Emrich, F., Hensel, J., Giesl, J.: AProVE: Modular termination analysis of memory-manipulating C programs. CoRR, abs/2302.02382 (2023). http://dx.doi.org/10.48550/arXiv.2302.02382
12. Frohn, F., Giesl, J.: Proving non-termination via loop acceleration. In: Barrett, C.W., Yang, J. (eds.) FMCAD 2019, pp. 221–230 (2019). http://dx.doi.org/10.23919/FMCAD.2019.8894271

13. Frohn, F., Giesl, J.: Proving non-termination and lower runtime bounds with LoAT (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning. IJCAR 2022. LNCS, vol. 13385, pp. 712–722. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_41
14. Giesl, J., et al.: Analyzing program termination and complexity automatically with AProVE. J. Autom. Reason. **58**(1), 3–31 (2017). http://dx.doi.org/10.1007/s10817-016-9389-x
15. Giesl, J., Rubio, A., Sternagel, C., Waldmann, J., Yamada, A.: The termination and complexity competition. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 156–166. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_10. http://termination-portal.org/wiki/Termination_Competition_2022
16. Hensel, J., Giesl, J., Frohn, F., Ströder, T.: Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution. J. Log. Algebr. Methods Program. **97**, 105–130, (2018). http://dx.doi.org/10.1016/j.jlamp.2018.02.004
17. Hensel, J., Mensendiek, C., Giesl, J.: AProVE: non-termination witnesses for C programs (competition contribution). In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13244, pp. 403–407. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_21
18. Hensel, J., Giesl, J.: Proving termination of C programs with lists. CoRR, abs/2305.12159 (2023). http://dx.doi.org/10.48550/arXiv.2305.12159
19. Le, T.C., Qin, S., Chin, W.N.: Termination and non-termination specification inference. In: Grove, D., Blackburn, S.M. (eds.) PLDI 2015, pp. 489–498 (2015). http://dx.doi.org/10.1145/2737924.2737993
20. Malík, V., Martiček, Š, Schrammel, P., Srivas, M., Vojnar, T., Wahlang, J.: 2LS: memory safety and non-termination. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 417–421. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_24
21. Metta, R., Yeduru, P., Karmarkar, H., Medicherla, R.K.: VeriFuzz 1.4: checking for (non-)termination (competition contribution). In: Sankaranarayanan, S., Sharygina, N. (eds.) TACAS 2023. LNCS, vol. 13994, pp. 594–599. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30820-8_42
22. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Automated termination analysis of Java Bytecode by term rewriting. In: Lynch, C. (ed.) RTA 2011, LIPIcs, vol. 6, pp. 259–276 (2010). http://dx.doi.org/10.4230/LIPIcs.RTA.2010.259
23. Richter, C., Wehrheim, H.: PeSCo: predicting sequential combinations of verifiers. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 229–233. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_19
24. Rowe, R.N.S., Brotherston, J.: Automatic cyclic termination proofs for recursive procedures in separation logic. In: Bertot, Y., Vafeiadis, V. (eds.) CPP 2017, pp. 53–65 (2017). http://dx.doi.org/10.1145/3018610.3018623
25. Ströder, T., et al.: Automatically proving termination and memory safety for programs with pointer arithmetic. J. Autom. Reason. **58**(1), 33–65 (2017). http://dx.doi.org/10.1007/s10817-016-9389-x
26. Zhao, J., Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations. In: Field, J., Hicks, M. (eds.) POPL 2012, pp. 427–440 (2012). http://dx.doi.org/10.1145/2103656.2103709

Reasoning About Regular Properties: A Comparative Study

Tomáš Fiedor, Lukáš Holík(✉), Martin Hruška, Adam Rogalewicz, Juraj Síč, and Pavol Vargovčík

Brno University of Technology, Brno, Czech Republic
{ifiedortom,holik,ihruska,rogalew,sicjuraj,ivargovcik}@fit.vutbr.cz

Abstract. Several new algorithms for deciding emptiness of Boolean combinations of regular languages and of languages of alternating automata have been proposed recently, especially in the context of analysing regular expressions and in string constraint solving. The new algorithms demonstrated a significant potential, but they have never been systematically compared, neither among each other nor with the state-of-the art implementations of existing (non)deterministic automata-based methods. In this paper, we provide such comparison as well as an overview of the existing algorithms and their implementations. We collect a diverse benchmark mostly originating in or related to practical problems from string constraint solving, analysing LTL properties, and regular model checking, and evaluate collected implementations on it. The results reveal the best tools and hint on what the best algorithms and implementation techniques are. Roughly, although some advanced algorithms are fast, such as antichain algorithms and reductions to IC3/PDR, they are not as overwhelmingly dominant as sometimes presented and there is no clear winner. The simplest NFA-based technology may sometimes be a better choice, depending on the problem source and the implementation style. We believe that our findings are relevant for development of automata techniques as well as for related fields such as string constraint solving.

1 Introduction

Efficient representation of regular properties of finite words has been the subject of research for a long time, with applications and results spanning much of the field of formal reasoning, including regular expression matching, verification, testing, modelling, or general decision procedures of logics. When regular properties are combined using Boolean and similar operations, interesting decision problems are PSPACE-complete. This includes the most essential problem of language emptiness (further just emptiness). The textbook approaches that use deterministic automata are plagued by state space explosion. Determinization and complementation is done by exponential subset construction and conjunction is quadratic. This motivated the research on efficient algorithms for non-deterministic and alternating finite automata (NFA and AFA, respectively).

Using nondeterminism and alternation, one can gain one or two levels of exponential savings in the size of automata, respectively. Alternation in context of automata was first studied in [24] and [18,38,53], and extensively in the context of automata over

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 286–306, 2023.
https://doi.org/10.1007/978-3-031-38499-8_17

infinite words and temporal logics (e.g., [57,58,66,76]). It adds conjunctive branching to the disjunctive non-deterministic branching and allows to avoid the blow-up in the automata size completely. However, from the perspective of the worst case complexity, the gained succinctness is payed back by the PSPACE-completeness of language emptiness. Still, the more succinct the representation gives more opportunities for clever heuristics that combat the worst case complexity and work in practical cases, essentially by avoiding re-creation of the entire (non)deterministic representation.

Several very promising techniques and their implementations were proposed during the recent years. The latest advances in testing AFA emptiness appeared in the context of analysing combinations of regular expressions and in string solving. A group of these techniques is based on reducing AFA emptiness to a reachability in a Boolean transition systems and using existing implementations of model-checking algorithms, most notably of IC3/PDR [15,46], such as ABC [17], nuXmv [22], or IC3Ref [16], to solve it [27,28,47,80]. The most recent contribution from [73] extends the SMT-solver Z3 with symbolic derivatives, a generalisation of Antimirov derivatives of regular expressions. Z3 uses them to convert a combination of regular expressions into an alternating/Boolean automaton and on the fly tests its language emptiness through the classical de-alternation and a search for an accepting configuration.

Slightly older algorithm for testing equivalence of AFA (convertible to an emptiness test) is based on computing bisimulation up-to congruence [30]. It generalizes the original NFA-equivalence test of [11]. The congruence closure algorithms were preceded by the antichain algorithms that optimize the subset construction by the subsumption pruning [41,82], and by the first attempt to use the model checking algorithms, namely the algorithm Impact of [63], to emptiness of combinations of regular properties [40]. Lastly, the area of string constraint solving gave rise to a large variety of string constraint solvers. They approach combinations of regular properties through a spectrum of clever techniques based e.g. on automata, transformations to other types of constraints, reasoning on lengths of strings, Parikh images, etc. (e.g. Z3 [65,73], CVC4/5 [7,68], Z3Str4 [9], OSTRICH [25,26], Trau [4,5] to name a few).

These works demonstrate a significant promise, but they are presented in specific, often narrow contexts and under varying views on state of the art. Consequently, they have never been sufficiently compared against each other. Even comparisons against the most efficient implementations of the more standard techniques based on (non)deterministic automata is rare. String solvers were compared only against string solvers, advanced AFA-emptiness tests were compared only against the basic de-alternation. A somewhat interesting comparison was done only between NFA-antichain and up-to congruence-based language inclusion and equivalence test in [11] and in [39], and between the basic antichain based AFA emptiness and a version that uses abstract interpretation [41]. A number of works also take as their baseline implementations of automata or string solvers which, even though being respectable tools in their own right, are currently not the fastest solvers of combinations of regular properties in either category. On top of that, all the mentioned works on solving combinations of regular properties use only narrow benchmarks, often mutually exclusive.

Systematic comparisons of tools and algorithms on meaningful benchmarks is obviously needed to answer the questions ‘What to use?’ and ‘What to compare with?’, and generally for the field of reasoning about regular properties and automata to progress. We thus present a comparison of implementations of major algorithms. We compare the tools on a large benchmark of problems that we have collected from other works, from string constraint solving problems, analysis of regular expressions, regular model checking, and analysing LTL properties of systems. We believe that it is currently the most comprehensive benchmark in existence. Our main focus is on examples around string solving and analysis of regular expressions, which is also where the most of the recent developments has happened. These benchmarks mostly allow for a relatively simple representations of automata transition functions. Even though the alphabets in examples coming form this are large (e.g. UNICODE with up to 2^{32} symbols), the alphabet size can, in most cases, be reduced to few symbols by working with alphabet minterms (classes of indistinguishable symbols) instead of individual symbols. The issue of effective symbolic representation of transition relations with large alphabets then does not dominate the evaluation, although it would be critical in other application areas, such as deciding WS1S (monadic second-order logic of one successor) or linear integer arithmetic [20,44,81].

We have obtained results that paint the basic landscape of the available techniques and tools. They identify tools and approaches which are likely to work well and should be used as the baseline in comparisons. We also provide a relatively diverse and large benchmark to be used in comparisons. The results broadly confirm that the new algorithms represent a leap in efficiency compared to the technology of DFA and also make a reduction of a problem to language emptiness of alternating automaton an attractive option. On the other hand, they challenge some folklore knowledge and conclusions implied elsewhere. For instance, reductions to IC3/PDR, although yielding one of the fastest algorithm, are not as vastly superior as sometimes presented. Some practically relevant benchmark categories are best solved by a combination of an antichain algorithm with a SAT solver. Others, surprisingly many in fact, by a simple efficiency oriented implementation of basic algorithms for nondeterministic automata. Our results also underscore that there is no universal silver bullet. The particular kind of the problem, determined to a large degree by its source, is a decisive factor that should be taken into account when choosing and tuning a solver.

We will maintain and further grow the benchmark set, at GitHub [1], as well as the framework for the entire comparison, at [2], in order for it to be easily usable and extensible by others.

2 Preliminaries

A *(nondeterministic) finite automaton (NFA)* over Σ is a tuple $\mathcal{A} = (Q, \Delta, I, F)$ where Q is a finite set of *states*, Δ is a set of *transitions* of the form $q \xrightarrow{a} r$ with $q, r \in Q$ and $a \in \Sigma$, $I \subseteq Q$ is the set of *initial states*, and $F \subseteq Q$ is the set of *final states*. A *run* of $\mathcal{A}$ over a word $w \in \Sigma^*$ is a sequence $p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} p_n$ where for all $1 \leq i \leq n$, it holds that $a_i \in \Sigma \cup \{\epsilon\}$, $w = a_1 \cdot a_2 \cdots a_n$, and either $p_{i-1} \xrightarrow{a_i} p_i \in \Delta$ or $p_{i-1} = p_i$, $a_i = \epsilon$. The run is *accepting* if $p_0 \in I$ and $p_n \in F$, and the *language* $L(\mathcal{A})$ of $\mathcal{A}$ is the set of all words for which $\mathcal{A}$ has an accepting run.

The automaton is *deterministic (DFA)* if for every state q and symbol a, Δ has at most one transition $q \xrightarrow{a} r$. Any NFA can be determinized by the *subset construction*, which creates the DFA $A' = (2^Q, \Delta', \{I\}, \{S \mid S \cap F \neq \emptyset\})$ where $S \xrightarrow{a} S' \in \Delta'$ iff $S' = \{s' \mid s \in S \wedge s \xrightarrow{a} s' \in \Delta\}$. The basic automata constructions implementing Boolean operations with languages are intersection, $\mathcal{A}\cap\mathcal{A}' = (Q\times Q', \Delta^{\times}, I\times I', F\times F')$ where $(q,q')\xrightarrow{a}(r,r') \in \Delta^{\times}$ iff $q\xrightarrow{a}r \in \Delta$ and $q'\xrightarrow{a}r' \in \Delta'$, non-deterministic union $\mathcal{A}\cup\mathcal{A}' = (Q\cup Q', \Delta\cup\Delta', I\cup I', F\cup F')$, deterministic union by product which is the same as $\cap$ up to that the final states are $F\times Q\cup Q\times F$, and complementation which consists of determinization and complementing the final states.

Alternating Automata. An *alternating finite automaton (AFA)* in the most general form would be a tuple $\mathcal{M} = (\Sigma, \mathbb{P}, Q, \delta, I, F)$ where, when denoting $\mathbb{B}(X)$ the Boolean predicate formulae over variables X: 1) Σ is a finite alphabet; 2) $\mathbb{P}$ is a set of unary *symbol predicates* with a free variable α; 3) Q is a finite set of *states*; 4) $\delta : Q \to \mathbb{B}(Q\cup\mathbb{P})$ is a *transition function* where states of Q have only positive occurrences 5) $I \in \mathbb{B}(Q)$ is a positive *initial condition*; and 6) $F \in \mathbb{B}(Q)$ is a negative *final/accepting condition*.[1]

It can be interpreted as the *forward NFA* $A^{\mathsf{f}} = (\Sigma, \mathcal{P}(Q), \Delta^{\mathsf{f}}, I', F')$ with states $c \subseteq Q$ called *configurations* of A. Assume many sorted interpretation of formulae over variables Q of the type Boolean (values 0 and 1) and the variable α of the type Σ. A set of states $c \subseteq Q$ is understood as an assignment $Q \to \{0,1\}$ in which $c(q) = 1$ corresponds to $q \in c$. A pair (c,a), $a \in \Sigma$ is understood as the same assignment extended with $\alpha \mapsto a$. The satisfaction relation $\models$ between a formula and a configuration c or a pair (c,a) is defined as usual. The transition relation Δ^{f} then contains a transition $c\xrightarrow{a}c'$ iff $(c',a) \models \bigwedge_{q\in c}\Delta(q)$, and I' and F' are the sets of configurations that satisfy I and F, respectively. It is common to define Δ^{f} to contain only the smallest transitions, that is, for a given c and a, only the transitions $c\xrightarrow{a}c'$ with the $\subseteq$-minimal target c' are in Δ.[2] The language of A, $L(A)$, is the language of A^{f}.

The AFA can equivalently be interpreted as the *backward NFA*, the automaton $A^{\mathsf{b}} = (\Sigma, \mathcal{P}(Q), \Delta^{\mathsf{b}}, I', F')$ where $c\xrightarrow{a}c' \in \Delta^{\mathsf{b}}$ if $(c,a) \models \Delta(q)$ for each $q \in c$. Here it is enough to take, for a given c' and a, only the transition with the $\subseteq$-largest source c[3] (this makes the transition relation backward deterministic).

Boolean Automata. Alternating automata may be extended to Boolean finite automata (BFA) by allowing any Boolean combination in the initial, final, and transition formulae (states in the initial and transition formulae may occur negatively, states in the final formula may occur positively). Note that the extension of AFA to BFA is not dramatic, as a BFA is easily encoded as an AFA with only double the size, by the following steps: 1) for each $q \in Q$, add state $\bar{q}$ with $\Delta(\bar{q}) = \neg\Delta(q)$, 2) transform all formulas in I, F, Δ to DNF, 3) replace all literals $\neg q$ by $\bar{q}$ in Δ and I and replace literals q by $\neg\bar{q}$ in F.

Restricted Forms of AFA Transition Relation. The general form of AFA, as defined above, is the most succinct. It provides space for most optimizations, such as in [77]. Automata in this form are generated from LTL conversions of [34] used in [30,77]. On the other hand, only a small subset of algorithms and tools support AFA in this most liberal form. A common restriction (used e.g. in [30]) is to separate symbols from states in

[1] This is not a most standard definition of AFA but it allows us to later cover and categorize their common syntactic variants. See e.g. [18,41,57] for more standard definitions.

[2] A state in a configuration is understood as a constraint. The less constraints, the more can be accepted from the configuration. Transitions to more constrained configurations are useless.

[3] Going backward, larger configurations are more permissive. Transitions from the same target with smaller configurations are useless.

the transition formulae, that is, having $\Delta(q)$ in the form $\varphi \wedge \psi$ with $\varphi \in \mathbb{B}(\mathbb{P}), \psi \in \mathbb{B}(Q)$. We call such AFA *separated*. The transition relation can then be seen as a function $Q \rightarrow \mathbb{B}(\mathbb{P}) \times \mathbb{B}(Q)$. Separated AFA are often considered with the state formula ψ in the disjunctive normal form (e.g. in [36,41]), which we call the *DNF form*, and Δ then may be seen as a set of transitions of the form $q \text{-}\{\varphi\}\!\!\rightarrow c$ where $\bigwedge c$ is a (positive) clause of ψ.

The Decision Problems. We will concentrate on two decision problems:

(1) *AFA emptiness* asks whether the language of the given AFA is empty.
(2) *Emptiness of Boolean combinations of regular properties* (*BRE*), asks whether a Boolean combination of regular languages, given as automata or regular expressions, is empty (languages can be combined with ∩, ∪, and complement wrt. Σ^*, which also covers testing inclusion and equivalence[4]).

3 Existing Algorithms and Tools

In this section, we will overview the existing approaches and tools implementing AFA and BRE emptiness.

3.1 Representation of Automata Transition Relations

In the simplest form, a predicate on a automata transition represents a single letter from the alphabet. This is called an *explicit transition*. Explicit automata are simple, allow for low level optimizations, and implementation of complex algorithms for them is manageable (such as advanced algorithms for computing simulations [23,50,70]). The technique of a-priori mintermization, that replaces the alphabet by the alphabet of minterms, classes of indistinguishable symbols, makes explicit automata usable also when alphabets are large. However, when the number of minterms tends to explode, explicit automata do not scale.

Various implementations of automata have been using transition predicates implemented as BDDs, Boolean formulae, formulae over SMT-theory of bit-vectors, intervals of numbers, etc. This has been systematized in the works on *symbolic automata* [31,33,79], where the symbol predicates may be taken from any effective Boolean algebra (and the automata are in the separated form). Even more compact than symbolic automata are representations of the transition relation used in the WS1S solver Mona or in some of the implementations of AFA, which in a way drop the restriction to the separated form. We will discuss the concrete implementations below.

3.2 (Non)deterministic Finite Automata

The baseline approach to solve BRE is to use DFA or NFA. Boolean operations are implemented as the classical construction listed in Sect. 2. Automata may be kept deterministic, or they are kept non-deterministic whenever possible and determinized only before complementing. An important ingredient of achieving efficiency is usually to

[4] $L' \subseteq L$ is emptiness of $L' \cap \overline{L}$ and equivalence is emptiness of $(L' \cap \overline{L}) \cup (\overline{L'} \cap L)$.

minimize automata at least once every few operations (important e.g. in applications such as regular model checking [12] or some approaches to string solving [4,10,25]). The deterministic approaches construct the minimal DFA by the Hopcroft, Moore, Brzozowski, or the Huffman algorithm [19,52,54,64], the non-deterministic approach may use simulation [23,45,50,55,70] or bisimulation [48,69,75] based reduction methods. Simulation reduces significantly more but is much costlier. DFA/NFA are implemented in many libraries. Here we select a representative sample.

First, ENFA is the simplest tool, our own implementation of NFA, which was originally meant to play the role of a baseline. It uses explicit automata with mintermization. It is implemented in C++, with efficiency in mind, but with no extensive optimizations (roughly, transitions from a state stored in a two layered data structure, the first layer divided and ordered by symbols, and the second layer ordered by the target state). It uses an off the shelf implementation of one of the newest generation algorithms for computing simulation [23,50,70] (that achieve good efficiency through a usage of the partition-relation data structure) taken from VATA tree automata library [59] (implementing namely [50]).[5]

The BRICS automata library [67] is often considered a baseline in comparisons [67]. It uses primarily deterministic automata and transition relation represented symbolically using character ranges. It is written in Java and relatively optimized.

The AUTOMATA library [78], made in C#, implements symbolic NFA/DFA parametrized by an effective Boolean algebra. We use it with the default algebra of BDDs. AUTOMATA has been long developed and has accumulated many optimizations and novel techniques for handling symbolic automata (e.g., optimized minimization [32]).

MONA [44], written in C, is the most influential and optimized implementation of deterministic automata. It specialises in deciding WS1S formulae, which besides Boolean combinations includes also quantification. The decision procedure generates DFA with complex transition relations over large alphabets of bit-vectors. For this purpose, MONA uses a compact representation of the transition relation: a single MTBDD for all transitions originating in a state, with the target states in its leaves. MONA can represent only a DFA, hence it always implicitly determinizes.

VATA [59], written in C++, is a library implementing non-deterministic tree automata. As NFA are a special case of tree automata, we can use it as an implementation of the basic constructions for explicit NFA. It is relatively optimized. We include it into the comparison for its fast implementation of the antichain inclusion checking [12,49], which for NFA boils down to the inclusion check of [36].

3.3 Alternating Automata

De-alternation. The basic approach to AFA emptiness is *de-alternation*, transformation to an NFA, either the forward A^{f} or the backward A^{b}, followed by testing the emptiness of the resulting NFA. Both NFAs are constructed by a variation on the NFA subset construction. We are not aware of any tool using pure de-alternation, and we believe that it would not be competitive. The forward algorithm is however the basis of [73]

[5] In our experiment, simulation is only used after parsing and has minimal overall impact.

used in Z3 where it is run on the fly with a novel symbolic derivative construction (discussed also in the paragraph on string constraint solvers).

Interpolation Based Abstraction Refinement. Attempts to harness model checking algorithms to AFA emptiness appeared in the context of string solving and processing of regular expressions. To our best knowledge, the earliest attempt was [40], where conjunctions of regular constraints were solved using the interpolation-based algorithm of [62]. The interpolation-based abstraction refinement, namely the algorithm Impact of [63], was also used in [56]. This work concentrated on more general problem, solving emptiness of AFA over data words with an infinite data domain (that can relate past and current values of data variables). Their tool JaltImpact [3] (in Java), that we include into our comparison, can be run on our benchmark too.

Reduction to Reachability and IC3/PDR. The work of [80] presented the first translation of string constraints (mostly BRE) into reachability in a Boolean transition system (circuit) that was then solved by the model checker nuXmv [22]. This was de facto the first reduction of AFA emptiness to reachability in a Boolean transition system (BTS).

Let us briefly overview the basic principle of the reduction. The *forward BTS* for an AFA A has configurations that are Boolean assignments to Q, initial and final configurations satisfy I and F, respectively, and transitions are given by the formula $\Phi^{\mathsf{f}}_{\Delta} : \bigwedge_{q \in Q} q \rightarrow [\Delta(q)]'$. Here we use $[\varphi]'$ to denote the formula obtained from φ by substituting every state q by its primed version q', and we will also denote by $[c]'$ the primed version $\{q' \mid q \in c\}$ of a configuration c. A *successor* of a configuration c is any configuration $\bar{c}$ such that $[\bar{c}]'$ satisfies $\exists Q \exists \alpha\, \Phi^{\mathsf{f}}_{\Delta} \wedge \bigwedge_{q \in C} q$ (the symbol variable alpha is of the bit-vector sort). *Reachability* is then the transitive and reflexive closure of the successor relation and the *reachability problem* asks whether a final configuration is reachable from an initial one. It is the case if and only if A is not empty. The forward reduction has been used in [80]. Alternatively, the *backward BTS* for A has the initial configurations satisfying F, final configurations satisfying I, and the successor relation given by the formula $\Phi^{\mathsf{b}}_{\Delta} : \bigwedge_{q \in Q} q' \rightarrow \Delta(q)$.

The work [28] applied IC3/PDR [15,46], implemented in IC3Ref [16], together with the backward BTS reduction to solve emptiness of BRE and obtained very encouraging results. The implementation used in [28], called Qzy, is, however, proprietary and not publicly available. Similar approach was taken by [47], where a string constraint was translated to a multi-tape AFA and then to a BTS by the forward translation, and given to IC3/PDR to solve through tools nuXmv [22] or ABC [17]. Results of [77] seem to indicate that the backward translation is better and the same is suggested by the comparison in [27,28] in which the string solver Sloth [47], based on the forward reduction, was much slower than Qzy, based on the backward reduction. In this comparison, we include our own C++ implementation bwIC3 of the backward reduction based on the model checker ABC.

Antichains. Antichain algorithms presented in [82] were the first breakthrough in solving BRE. They use subsumption relations between the states of the automata constructed by variations of the subset construction to prune the constructions. They were used to test language universality and inclusion of NFAs and AFA emptiness. The AFA emptiness namely is based on an on-the-fly search for an accepting state of the A^{f} or for an initial state of the A^{b}. Subsumption prunes discovered states that are larger (smaller for the backward algorithm) than others.

The antichain algorithms were enhanced and generalized in a number of works, e.g. with a more aggressive pruning by the simulation-based subsumption [6,36], or by counterexamples guided abstraction refinement in [41]. In this comparison, we include the NFA inclusion check implemented in the VATA tree automata library [59]. We also experimented with a student-made implementation of the antichain AFA emptiness check of [41] that uses abstraction refinement (the original implementation is no longer maintained and we were not able to run it). However, not being able to achieve a competitive performance, we excluded it from the comparison. One reason of the poor performance may be that simplest form of AFA, explicit DNF form (used in the original version [41]), might be too inefficient and costly to construct in our examples, partly due to a large number of minterms induced by the AFA emptiness benchmark.

We implemented (in C++) the antichain AFA emptiness test of [36] that integrates tightly with a SAT solver to handle the general form of AFA with large alphabets. We will refer to it as ANTISAT. We will briefly explain its principle. It essentially implements the reachability test for the backward BTS discussed in the previous paragraph. A configuration c is represented by the conjunction $\phi_c = \bigwedge_{q \in Q \setminus c} \neg q$. Note that ϕ_c is satisfied by the downward closure of c, which are all configurations included in (subsumed by) c. To compute predecessors of configurations represented by ϕ_c, the SAT solver (namely MiniSAT [37]) is called on the formula $\Phi : \Psi^{\mathsf{b}}_{\Delta} \wedge \phi_c \wedge \psi_{\mathsf{Ach}}$. Here, ψ_{Ach} excludes all already discovered configurations from the solution. It is a conjunction of clauses $\overline{\phi_c} : \bigvee_{q \in Q \setminus c} q$ for every previously discovered configuration c. The SAT solver discovers a satisfying assignment e, which is turned into a new configuration $c' = Q \cap e$ (that is, the values of the symbol bits constituting the bit-vector α are omitted from e). Unless c' is initial, it is queued for further predecessor computation and is immediately added to ϕ_{Ach} through the interface of incremental SAT solving as the clause $\overline{\phi_{c'}}$. Finally, only maximal predecessors of c are of interest, as the non-maximal ones are subsumed by them. We enforce the maximality of c through working directly with the internal SAT solver structures: at decision points, the SAT solver is forced to give priority to decisions that assign 1 to state variables.

Bisimulation up-to Congruence. A later class of algorithms, here refered to as *up-to algorithms*, checks equivalence as a bisimulation between configurations of AFA, and utilises the up-to congruence technique to prune the search space. The first algorithm on NFA equivalence [11] was extended to alternating automata emptiness check in [30]. These algorithms are close to antichains. As shown in [11], the pruning potential of the up-to techniques is in theory the same or larger than that of antichain. A disadvantage of the up-to congruence technique is the need for expensive evaluation of congruence closures. The more extensive experiments of [39] shows antichain algorithms as faster, with an exception of randomly generated automata with small alphabets and very dense transition relations. We include into the comparison the Java implementation of the AFA-emptiness of [30] (emptiness reduces to equivalence with a trivial empty AFA), that we refer to as BISIM. The other implementations of up-to algorithms we are aware of, from [39] and [11], are single-purpose programs that decide equivalence of two NFAs, hence we would be able to run them on a very small fraction of our benchmark only.

3.4 String Constraints Solvers

There are dozens of string constraint solvers that implement, to a various degree, a support for deciding combinations of regular properties. String languages are rich and BRE are not the absolute priority of the solvers, hence they perform on them generally worse than specialised tools. However, string solvers implement a wide scale of unique techniques and pragmatic heuristics that may work in specific instances. Representatives of the solvers with the most mature implementations (also used in most comparisons in the literature) are Z3 [65,73] and CVC5 [7,68]. CVC5 solves BRE mostly through rewriting rules. Recently [73] extended Z3 with an approach based on the Antimirov derivative automata construction generalised to symbolic automata and extended regular expressions. Essentially, the construction produces a symbolic AFA/BFA and checks its emptiness on the fly while running the forward de-alternation. As shown in [73], it is significantly more efficient in solving BRE than other SMT solvers (including CVC5).

3.5 Other Approaches and Tools

Although we believe that we have collected a representative subset of existing algorithms and tools, we have not collected all interesting specimens. Some were not available, some were difficult to run or prepare the inputs for, some seemed covered by experimentation in other works. Including these tools and algorithms into the comparison could still be interesting and we leave it for the future work (we plan to keep extending the tool base as well as the benchmark set). Namely, the tool DPRLE [51], used in the comparison in [28], seemed to be mostly outperformed by the IC3/PDR approach implemented in Qzy, however, not absolutely consistently. The implementation of NFA antichain and up-to congruence techniques used in [39] seems efficient, with its NFA antichain inclusion twice as fast as that of VATA. The up-to congruence NFA equivalence checking of [11] could be fast too ([11] and [39] report somewhat conflicting results). There are numerous NFA/DFA libraries, e.g. the C alternative of BRICS [61] or the Java implementation of symbolic NFA of [29]. ALASKA [35] might contain interesting implementations of antichain algorithms but is no longer maintained and available. Our comparison is missing a basic implementation of antichain-powered de-alternation for explicit AFA in the DNF form, which, if not overwhelmed by a large number of minterms, could reach a good performance through simple fast data structures, similarly to our ENFA.

4 Benchmarks

We collected as comprehensive benchmark as possible, harvesting examples used in previous works as well as generating some of our own. It is available together with the whole experiment from [2] and at GitHub [1] (we plan to maintain and grow the benchmark and welcome contributors).

Our main focus of the current benchmark set is the areas where the most of the development in solving AFA and BRE emptiness happened recently, which is string constraint solving and analysis of regular expressions used in analysing and filtering texts. Atomic regular properties are here mostly given in the form of regular expressions over UNICODE character classes. The alphabet is large but the number of minterms is mostly small or moderate. This is true also for our examples from regular model checking. Symbolic handling of complex transition relations over large alphabets is thus not absolutely crucial and the experiment can stay focused on the main algorithms for emptiness check. For that reason, we do not include benchmarks from solving WS1S [21], the primary target of MONA, or Presburger arithmetic with automata [13,81], where the techniques of handling symbolic alphabet are indispensable. Techniques specialising at this kind of problems would deserve their own study. Our benchmarks where the symbolic alphabet representation is still rather important are AFA coming from (combinations of) LTL properties, with alphabets of sets of atomic propositions, and from translations of string constraint problems to AFA with complex multi-track alphabets.[6]

Boolean Combinations of Regular Expressions. This group of BRE contains benchmarks on which we can run all tools, including those based on NFA and DFA. They have small to moderate numbers of minterms (about 30 in average, at most over a hundred).

b-smt contains 330 string constraints from the Norn and SyGuS-qgen, collected in SMT-LIB benchmark [8], that fall in BRE. These were also used to compare SMT-solvers in [73].

b-hand-made has 56 difficult handwritten problems from [73] containing membership in regular expressions extended with intersection and complement. They encode (1) date and password problems, (2) problems where Boolean operations interact with concatenation and iteration, and (3) problems with exponential determinization.

b-armc-incl contains 171 language inclusion problems from runs of abstract regular model checking tools (verification of the bakery algorithm, bubble sort, and a producer-consumer system) of [12]. These examples were used also in [11,39].

b-regex contains 500 problems, obtained analogously as in [30,77], of the form $r_1 \wedge r_2 \wedge r_3 \wedge r_4 = r_1 \wedge r_2 \wedge r_3 \wedge r_4 \wedge r_5$, where each r_i is one of the 75 regexes[7] from RegExLib [71] selected so that $r_1 \wedge r_2 \wedge r_3 \wedge r_4 \wedge r_5$ is not empty. This benchmark is inspired by spam filtering, where we want to test whether a new filter r_5 adds anything to existing filters. We transformed this problem into the inclusion $r_5 \subseteq r_1 \wedge r_2 \wedge r_3 \wedge r_4$, and kept the original form for BISIM which expects an equivalence.

b-param has 8 parametric problems. Four are from [40]:

(1) `[a-c]a[a-c]`$\{n+1\} \cap$ `[a-c]a[a-c]`$\{n\}$ (long strings),

[6] We did not attempt to generate purely random problems. First, purely random automata generated e.g. by [74] seem to have different characteristics than automata coming from practical problems (e.g. in [12,39]). Second, although generating random NFA is possible with a generator controlled by three simple parameters which give a manageable parameter-value space covering all NFA, it is not clear how to similarly generate random AFA or BRE. On the other hand, we do include a benchmark based on randomly generated LTL formulae, which we consider relatively close to realistic LTL specifications.

[7] https://github.com/lorisdanto/symbolicautomata/blob/master/benchmarks/src/main/java/regexconverter/pattern%4075.txt.

(2) $\bigcap_{i=1}^{n}\texttt{([0-1]}\{i-1\}\texttt{0[0-1]}\{n-1\}\texttt{0[0-1]}\{n-i\}\alpha_i\texttt{)}\mid\texttt{([0-1]}\{i-1\}\texttt{1[0-1]}\{n-1\}\texttt{1[0-1]}\{n-i\}\alpha_i\texttt{)}$ (exponential branching),

(3) $\bigcap_{i=1}^{n}\texttt{.*(.}\{p_{10+i}\}\texttt{)+}\alpha_i$ (exponential paths 1), and

(4) $\bigcap_{i=1}^{n}\texttt{.+}\alpha_i\texttt{0(.}\{p_{10+i}\}\texttt{)+}$ (exponential paths 2), where $\alpha_1,\ldots,\alpha_n$ are disjoint character classes and p_j is the j-th prime number. Another four are from [28]:

(5) `^.[01]*.1.[01]{n}.$\^.[01]*.0.[01]{n-1}.$` (sat. difference),

(6) `^.[01]*.1.1.[01]{n}.$\^.[01]*.0.[01]{n+1}.$` (unsat. difference),

(7) `^.[01]*.1.[01]{n}.$∩^.[01]*.0.[01]{n-1}.$` (sat. intersection) and

(8) `^.[01]*.1.[01]{n}.$∩^.[01]*.0.[01]{n}.$` (unsat. intersection). For (1) we chose $n \in \{50, 100, \ldots, 500\}$, for (2)–(4) we chose $n \in \{2, 3, \ldots, 60\}$ and for (5)–(8) we chose $n \in \{50, 100, \ldots, 1000\}$.

AFA Benchmark. The second group of examples contains AFA not easily convertible to BRE. Here we can run only tools that handle general AFA emptiness. Some of these benchmarks also have large sets of minterms (easily reaching to thousands) and complex formulae in the AFA transition function, hence converting them to restricted forms such such as separated DNF or explicit may be very costly. This also seems to be the main reason for which our implementation of [41] could not compete.

a-ltlf-patterns comes from transformation of linear temporal logic formulae over finite traces (LTL$_f$) to AFA [34]. The 1699 formulae are from [60][8] and they represent common LTL$_f$ patterns which can be divided into two groups: (1) 7 parametric patterns (100 each) and (2) randomly generated conjunctions of simpler LTL$_f$ patterns (999 formulae).

a-ltl-rand contains 300 LTL$_f$ formulae obtained with the random generator of [77]. The generator traverses the syntactic tree of the LTL grammar, and is controlled by the number of variables, probabilities of connectives, maximum depth, and average depth. We have set the parameters empirically in a way likely to generate examples difficult for the compared solvers (the formulae have 6 atomic propositions and maximum depth 16).

a-ltl-param has a pair of hand-made parametric LTL$_f$ formulae (160 formulae each) used in [30,77]: *Lift* [43] describes a simple lift operating on a parametric number of floors and *Counter* [72] describes a counter incremented modulo the parameter.

a-ltlf-spec [60] contains 62 LTL$_f$ formulae that specify realistic systems, used by Boeing [14] and NASA [42]. The formulae represent specifications used for designing Boeing AIR 6110 wheel-braking system and for designing NASA NextGen air traffic control (ATC) system.

a-sloth 4062 AFA emptiness problems to which the string solver Sloth reduced string constraints [47]. The AFA have complex multi-track transitions encoding Boolean operations and transductions, and a special kind of synchronization of traces requiring complex initial and final conditions.

a-noodler 13840 AFA emptiness problems that correspond to certain sub-problems solved within the string solver Noodler in [10]. The AFA were created similarly as those of a-sloth, but encode a different particular set of operations over different input automata.

[8] https://drive.google.com/file/d/1eOYGvm3C8sQ-9iyfZ8qx42K54hgrFNTC.

5 The Comparison

We ran our experiments on Debian GNU/Linux 11, with Intel Core 3.4 GHz processor, 8 CPU cores, and 20 GB RAM. All experiments were run with the timeout of 60 s (increasing the timeout did not have a significant impact). Additional details as well as the virtual machine with the entire benchmark are available at [2].

Benchmarking Infrastructure. The initial difficulty is that the tools expect different input formats and forms of automata and the benchmarks come in different formats as well. We converted all benchmarks to our internal AFA format, from which we generated formats supported by the AFA handling tools JaltImpact, bwIC3, Antisat, and Bisim, or we extend the tools with a parser. The BRE benchmarks come from various sources. We first convert them into a master file which specifies the Boolean combination of atomic NFA, each atomic NFA stored in a separate file. The SMT-lib format is generated for Z3 and CVC5. In the case of b-hand-made, b-param, and b-smt, the atomic automata are translated from regular expressions using the parser of Brics, while in the case of b-regex, where the regexes contain features not supported by Brics, we use the parser from Bisim. b-smt and b-hand-made requires first translating from SMT-lib to a regular expression. In the case of b-armc-incl, the atomic automata come directly as NFAs, and are converted into formats of the individual BRE solvers (we again wrote parsers for some of the solvers), and to our AFA format for the AFA solvers. Every BRE solver was extended by an interpreter of the master file that reads the NFA/DFA from the generated solver-specific files (except the SMT solvers, which read SMT-lib). We note that due to some difficulties with internal structures, we currently cannot run Brics on b-armc-incl, and due to the lack of a converter from complex regular expressions and from pure NFA to the SMT format, we do not run Z3 and CVC5 on b-regex and on b-armc-incl.

Measured Data. We will present the results obtained with BRE (where we run all the tools) and with AFA emptiness (where we run bwIC3, Antisat, Bisim, and JaltImpact) separately. We also separate the results on examples from applications from results on parametric hand-made examples.

Table 1 summarizes the statistics from evaluating the benchmarks. The table lists: (i) the average time, (ii) the median time, and (iii) the number of timeouts and number of errors (mostly, a tool ran out of the memory, made a bad alloc or ran into a segmentation fault). A few errors, e.g. in CVC5 or Bisim, were due to the unsupported features in the inputs. The tools' performance is then visualised on cactus plots in Fig. 1. For each tool, the plot shows the progress of the tool on each benchmark: the y axis is the cumulative time taken on the benchmark, with the individual examples on the x axis ordered by the runtime taken by the tool. Timeouts are omitted. In the appendix, we also show a set of scatter-plots that compare for every benchmark the three best performing tools.

Finally, we compared the tools on the parametric benchmarks a-ltl-param and b-param. We illustrate the results in Fig. 2. Each graph shows the times for the increasing value of the specific parameter on the x axis.

Table 1. Summary of AFA and BRE benchmarks. Table lists (i) the average, (ii) the median, and (iii) the number of timeouts and errors (in brackets). Winners are highlighted in bold.

	a-ltl-rand			a-ltl-spec			a-ltlf-patterns			a-noodler			a-sloth			a-ltl-param		
		(300)			(62)			(1 699)			(13 840)			(4 062)			(320)	
BWIC3	**0.1**	**0.1**	**0**	0.1	0.1	0	**0.1**	**0.1**	**0**	**0.1**	**0.1**	**3**	**1.3**	**0.1**	**34**	**25.4**	**0.6**	**134**
BISIM	4.4	1.0	8	32.9	60.0	32	37.0	60.0	1013	31.6	26.4	6644(8)	17.5	1.5	1087(10)	58.2	60.0	308
JALTIMPACT	7.9	2.3	12	2.4	1.4	0(1)	4.0	2.8	0	3.8	1.8	186	24.1	15.4	958	47.0	60.0	205
ANTISAT	18.3	0.1	84	**0.0**	**0.0**	**0**	31.0	60.0	868	0.4	0.0	57	14.9	0.0	991	58.3	60.0	310

	b-armc-incl			b-hand-made			b-regex			b-smt			b-param		
		(171)			(56)			(500)			(330)			(267)	
BWIC3	5.2	1.1	1	0.4	0.1	0	**0.2**	**0.1**	**0**	0.1	0.1	0	44.9	60.0	191
BISIM	28.5	9.5	72	11.2	1.0	8	3.8	1.3	15	2.5	2.5	0	55.4	60.0	240
BRICS		-		3.9	0.4	3	5.8	0.8	40	0.3	0.3	0	52.7	60.0	228
CVC5		-		27.4	0.8	10(15)		-		0.8	0.2	1	48.6	60.0	208
AUTOMATA	3.5	0.4	9	0.2	0.2	0	0.2	0.2	0	0.2	0.2	0	46.3	60.0	161(42)
JALTIMPACT	30.9	24.6	63	11.1	3.6	5	12.2	2.4	48	3.5	3.5	0	57.8	60.0	252
ANTISAT	42.8	60.0	118	1.4	0.0	1	9.3	1.4	45	0.0	0.0	0	39.0	60.0	147
MONA	28.5	44.1	43	27.3	0.1	22(3)	41.0	60.0	15(298)	1.5	0.0	8	44.9	60.0	25(169)
ENFA	**1.9**	**0.8**	**0**	**0.1**	**0.0**	**0**	**0.2**	**0.1**	**0**	**0.0**	**0.0**	**0**	44.6	60.0	143(51)
VATA	2.6	3.4	0	**0.1**	**0.0**	**0**	2.1	0.2	10(1)	**0.0**	**0.0**	**0**	37.8	60.0	155(1)
Z3		-		3.9	0.0	2		-		0.4	0.0	2	**32.0**	**48.1**	**129**

5.1 Discussion

Based on the measurements, we make several observations.

Firstly, the tool which combines universality (it can be run on AFA as well as on BRE emptiness) with the most consistent good performance is BWIC3. It dominates most of the AFA emptiness benchmark, shows great or a very good performance on the BRE benchmark, and often stands out on the parametric examples. Moreover, the measurements reported in [28] suggest that the backward BTS reduction has even more potential. This is visible namely from the comparison of our results on the parametric benchmarks diff-sat, diff-unsat, inter-sat, and inter-unsat. Our implementation matched the result of [28] on diff-sat and partially on inter-sat, saw a worse trend on diff-unsat and much worse trend on inter-unsat. A likely culprit is a different underlying model-checker, ABC [17] in our implementation versus IC3Ref [16] in [28]. However, IC3Ref was not used out of the box in [28], harnessing it efficiently for problems of our king is not entirely trivial.

Secondly, the results on application related BRE (all BRE except the parametric examples in b-param) quite surprisingly favour the tools based mostly on relatively basic NFA algorithms. The overall best is the simplest tool of all, our implementation ENFA of basic NFA constructions. Close to the performance of ENFA is VATA, which uses the antichain inclusion checking on b-armc-incl and b-regex (the fact that explicit complementation of ENFA is faster than the antichain of VATA suggests that the inclusion benchmarks are not particularly hard). VATA specialises to the more general tree automata, which probably causes unnecessary overhead. AUTOMATA also performs well. It uses slightly more advanced algorithms than ENFA (such as lazy evaluation of difference, though, without antichain pruning). Its symbolic representation of transition functions with BDDs probably does not provide much advantage here. This result challenges the view that translating complex problems, arising for instance in string con-

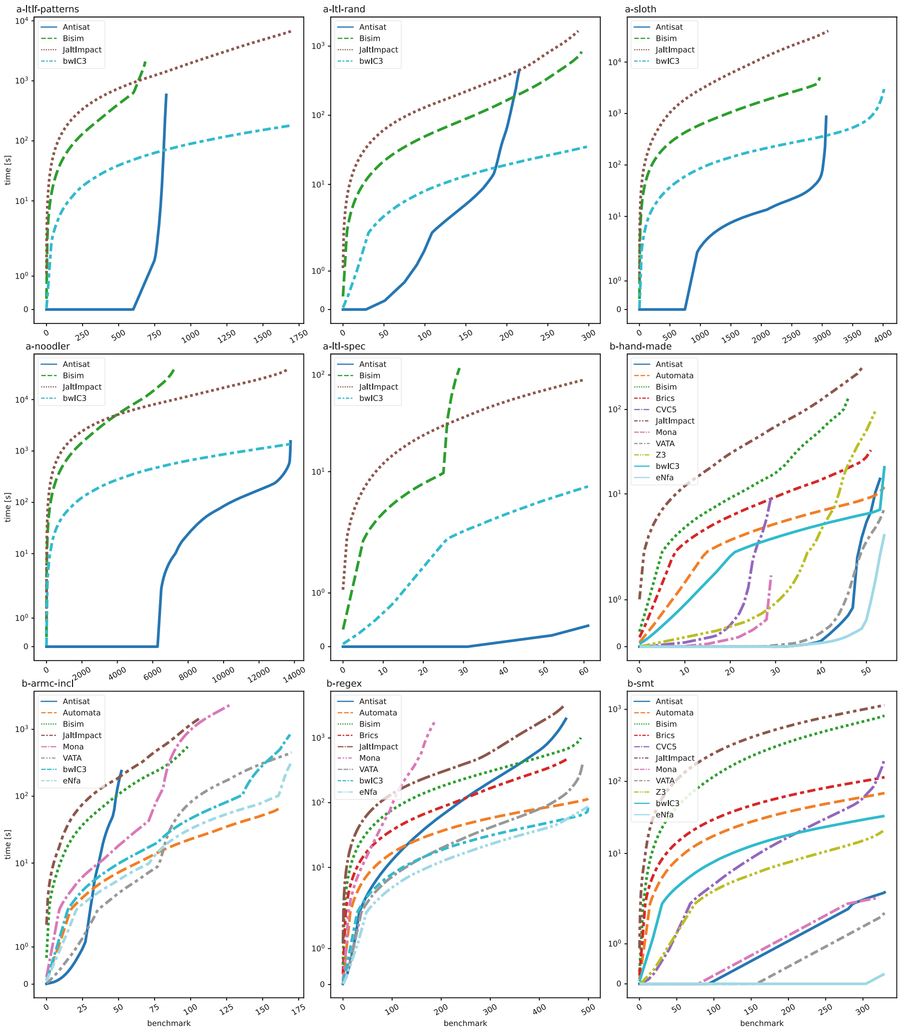


Fig. 1. Cactus plots of AFA and BRE benchmarks. The y axis is the cumulative time taken on the benchmark in logarithmic scale, benchmark on the x axis are ordered by the runtime of each tool.

straint solving, into AFA in order to use the sophisticated machinery of AFA solvers is an obvious silver bullet. Organizing the computation into smaller NFA operations, where, moreover, partial results can be minimized and re-used, and a simpler and hence more flexible NFA technology is used, might be a better strategy (this seems to work very well for instance in our recent prototype string constraint solver [10]).

Our AFA emptiness test ANTISAT based on the antichain algorithm and a SAT solver has an interesting performance. As can be seen on the cactus plots, besides its absolute domination on a-ltlf-spec, it is significantly faster than other tools on a large

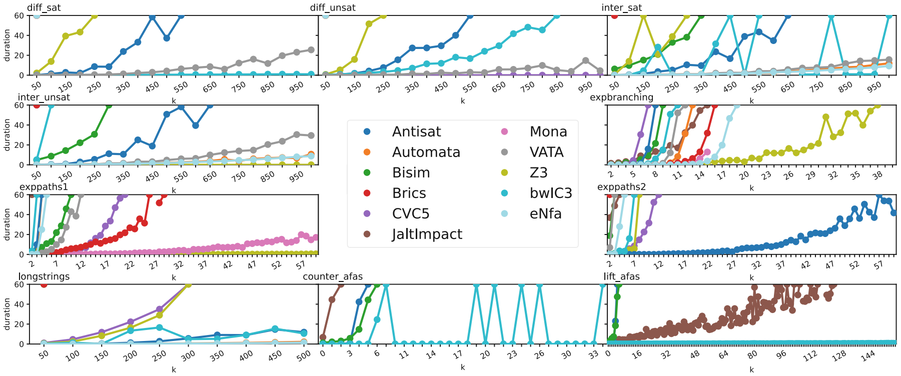


Fig. 2. Models of runtime on parametric benchmarks based on specific parameter k with timeout 60s. The sawtooths represent the tool failed on the benchmark for some k while solving benchmarks for $k-1$ and $k+1$. For brevity, we draw the models only until they start continually failing.

portion of the other AFA emptiness benchmark, but struggles on the rest. The examples where it dominates are often automata with the structure resembling a lasso (or several lassos) with a long handle. The other implementation of an antichain algorithm, NFA/NTA inclusion in VATA, also shows a good performance. This together points on the overall strength of antichain algorithms.

The SMT string constraint solvers are not among the best in the benchmark related to practical applications, but are competitive (especially Z3), and win on some parametric cases. This may be due to that various heuristics unique to SMT solvers, especially rewriting that reduces one type of a constraint to another, kicks in. For instance, Z3 seems to solve exppaths1 with a help of rewriting to the sub-string constraint in the theory of sequences. In general, the measurements on parametric examples underscore the fact that no algorithm is universally the best and their relative performance may vary drastically depending on the kind of an input.

Although the mediocre performance of the other tools can be partially explained by their focus on a different kind of a problem or a dated underlying technology, and each of them is respectable in its own right, a point can be made against relying on them as a baseline in comparisons of tools for solving our kind of problem. MONA, optimized for a different settings (complex alphabets of bit-vectors with many minterms), is held back by the implicit determinization, and, in our case, probably by the overhead of the symbolic representation. It also frequently runs out of the 32-bit address space for BDD nodes. Similarly for BRICS, which also always determinizes. The low performance of BISIM is surprising relative to the good results of the up-to algorithms reported in [11,30]. It is more consistent with [39] where up-to algorithms were not wining against antichains on the more practical examples. Our results however do not directly contradict the results of [30] itself, since it does not compare with the fast tools identified here and stands to a large degree on parametric and random benchmarks. There is also always the possibility that we have prepared the input in a way not ideal for the tool. For instance, transformation to the separated AFA, required by BISIM, is not entirely trivial. Further investigation of this and a comparison with some other implementation of the up-to techniques seems to be needed. The lack of a raw speed of JALTIMPACT on BRE and AFA emptiness is expectable considering that it is meant for a different kind of systems, AFA over data words. The stable trends shown in the graphs suggest that an implementation of an interpolation-based abstraction refinement optimized for BRE and AFA emptiness might have a potential.

Main Takeaways. The backward reduction of AFA emptiness to BTS reachability in a combination with IC3 is very fast and extremely versatile, showing very good performance on almost all benchmarks. However, on BRE with a relation to a real world application, simple NFA algorithms actually tend to have the best raw performance, with the simplest implementation of NFA being the best. Antichain algorithms work also well, even significantly better than other algorithms on specific kinds of AFA. These seem to be the tools to use. Reasonable implementations of the backward BTS reduction with IC3, of antichain, and of basic NFA should also be the baseline of comparisons.

MONA and BRICS, based on DFA, as well as JALTIMPACT focused on data words rather then on pure regular properties, do no reach the performance of the best tools. Also BISIM did not confirm the power of up-to algorithms. SMT-solvers, Z3 especially, are competitive, but cannot be considered the top of state of the art.

Generally, the particular kind and source of benchmark is a decisive factor influencing the performance of tools, as especially visible on the parametric benchmark.

Threads to Validity. Our results must be taken with a grain of salt as the experiment contains an inherent room for error. Although we tried to be as fair as possible, not knowing every tool intimately, the conversions between formats and kinds of automata, discussed at the start of Sect. 5, might have introduced biases into the experiment. Tools are written in different languages and some have parameters which we might have used in sub-optimal way (we use the tools in their default settings), or, in the case of libraries, we could have used a sub-optimal combination of functions. We also did not measure memory peaks, which could be especially interesting e.g. in when the tools are deployed on a cloud. We are, however, confident that our main conclusions are well justified and the experiment gives a good overall picture. The entire experiment is available for anyone to challenge or improve upon [2].

Acknowledgments. This work has been supported by the Czech Ministry of Education, Youth and Sports ERC.CZ project LL1908, and the FIT BUT internal project FIT-S-20-6427.

References

1. The benchmark used in the paper. https://github.com/VeriFIT/automata-bench
2. Experiment replication package and additional material. https://www.fit.vutbr.cz/research/groups/verifit/tools/afa-comparison/
3. Jaltimpact. https://github.com/cathiec/JAltImpact
4. Abdulla, P.A., et al.: TRAU: SMT solver for string constraints. In: Proceedings of the FMCAD'18. IEEE (2018)

5. Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janků, P.: Chain-free string constraints. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 277–293. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_16
6. Abdulla, P.A., Chen, Y.-F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 158–174. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_14
7. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
8. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016). https://www.smt-lib.org/
9. Berzish, M.: Z3str4: a solver for theories over strings. Ph.D. thesis (2021). http://hdl.handle.net/10012/17102
10. Blahoudek, F., et al.: Word equations in synergy with regular constraints. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) Formal Methods. FM 2023. LNCS, vol. 14000, pp. 403–423. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-27481-7_23
11. Bonchi, F., Pous, D.: Checking NFA equivalence with bisimulations up to congruence. In: Proceedings of the POPL'13. ACM (2013)
12. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In: Ibarra, O.H., Ravikumar, B. (eds.) CIAA 2008. LNCS, vol. 5148, pp. 57–67. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70844-5_7
13. Boudet, A., Comon, H.: Diophantine equations, presburger arithmetic and finite automata. In: Kirchner, H. (ed.) CAAP 1996. LNCS, vol. 1059, pp. 30–43. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61064-2_27
14. Bozzano, M., et al.: Formal design and safety analysis of AIR6110 wheel brake system. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 518–535. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_36
15. Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: Proceedings of the FMCAD'07. IEEE (2007)
16. Bradley, A.: IC3 reference implementation: a short, simple, fairly competitive implementation of IC3 (2015). https://github.com/arbrad/IC3ref
17. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
18. Brzozowski, J., Leiss, E.: On equations for regular languages, finite automata, and sequential networks. Theor. Comput. Sci. **10**(1) (1980)
19. Brzozowski, J.A.: Canonical regular expressions and minimal state graphs for definite events. In: Proceedings of the Symposium on Mathematical Theory of Automata (1962)
20. Büchi, J.R.: Weak Second-Order Arithmetic and Finite Automata, pp. 398–424. Springer, New York, NY (1990). https://doi.org/10.1007/978-1-4613-8928-6_22
21. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Proceedings of the International Congress on Logic, Method, and Philosophy of Science. SUP (1962)
22. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
23. Cécé, G.: Foundation for a series of efficient simulation algorithms. In: Proceedings of the LICS'17. IEEE (2017)
24. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. J. ACM **28**(1) (1981)
25. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replaceall function. In: Proceedings of the POPL'18 (2018)

26. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. In: Proceedings of the POPL'19 (2019)
27. Cox, A.: Model Checking Regular Expressions (2019). presented at MOSCA'19. https://mosca19.github.io/slides/cox.pdf
28. Cox, A., Leasure, J.: Model checking regular language constraints. CoRR abs/1708.09073 (2017)
29. D'Anthoni, L.: A symbolic automata library. https://github.com/lorisdanto/symbolicautomata
30. D'Antoni, L., Kincaid, Z., Wang, F.: A symbolic decision procedure for symbolic alternating finite automata. Electron. Notes Theor. Comput. Sci. **336** (2018)
31. D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 47–67. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_3
32. D'Antoni, L., Veanes, M.: Minimization of symbolic automata. In: Proceedings of the POPL'14. ACM (2014)
33. D'Antoni, L., Veanes, M.: Minimization of symbolic tree automata. In: Proceedings of the LICS'16. ACM (2016)
34. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Proceedings of the IJCAI'13. ACM (2013)
35. De Wulf, M., Doyen, L., Maquet, N., Raskin, J.F.: Alaska. In: Cha, S., Choi, J.Y., Kim, M., Lee, I., Viswanathan, M. (eds.) Automated Technology for Verification and Analysis. ATVA 2008. LNCS, vol. 5311, pp. 240–245. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88387-6_21
36. Doyen, L., Raskin, J.-F.: Antichain algorithms for finite automata. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 2–22. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_2
37. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
38. Fellah, A., Jürgensen, H., Yu, S.: Constructions for alternating finite automata. Int. J. Comput. Math. **35** (1990)
39. Fu, C., Deng, Y., Jansen, D.N., Zhang, L.: On equivalence checking of nondeterministic finite automata. In: Larsen, K.G., Sokolsky, O., Wang, J. (eds.) SETTA 2017. LNCS, vol. 10606, pp. 216–231. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69483-2_13
40. Gange, G., Navas, J.A., Stuckey, P.J., Søndergaard, H., Schachte, P.: Unbounded model-checking with interpolation for regular language constraints. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 277–291. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_20
41. Ganty, P., Maquet, N., Raskin, J.: Fixed point guided abstraction refinement for alternating automata. Theor. Comput. Sci. **411**(38–39) (2010)
42. Gario, M., Cimatti, A., Mattarei, C., Tonetta, S., Rozier, K.Y.: Model checking at scale: automated air traffic control design space exploration. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 3–22. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_1
43. Harding, A.: Symbolic strategy synthesis for games with LTL winning conditions. Ph.D. thesis, University of Birmingham (2005)
44. Henriksen, J.G., et al.: Mona: monadic second-order logic in practice. In: Brinksma, E., Cleaveland, W.R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60630-0_5

45. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proceedings of the FOCS. IEEE (1995)
46. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_13
47. Holík, L., Janků, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. In: Proceedings of the POPL'18, vol. 2 (2018)
48. Holík, L., Lengál, O., Síč, J., Veanes, M., Vojnar, T.: Simulation algorithms for symbolic automata. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 109–125. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_7
49. Holík, L., Lengál, O., Šimáček, J., Vojnar, T.: Efficient inclusion checking on explicit and semi-symbolic tree automata. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 243–258. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_18
50. Holík, L., Šimáček, J.: Optimizing an LTS-simulation algorithm. Comput. Inform. **7**, 1337–1348 (2010)
51. Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: PLDI'09. ACM (2009)
52. Hopcroft, J.E.: An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA (1971)
53. Hromkovič, J.: On the power of alternation in automata theory. J. Comput. Syst. Sci. **31**(1) (1985)
54. Huffman, D.: The synthesis of sequential switching circuits. J. Franklin Inst. **257**(3) (1954)
55. Ilie, L., Navarro, G., Yu, S.: On NFA reductions. In: Karhumäki, J., Maurer, H., Păun, G., Rozenberg, G. (eds.) Theory Is Forever. LNCS, vol. 3113, pp. 112–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27812-2_11
56. Iosif, R., Xu, X.: Abstraction refinement for emptiness checking of alternating data automata. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 93–111. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_6
57. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Trans. Comput. Logic **2**(3) (2001)
58. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM **47**(2) (2000)
59. Lengál, O., Šimáček, J., Vojnar, T.: VATA: a library for efficient manipulation of non-deterministic tree automata. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 79–94. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_7
60. Li, J., Pu, G., Zhang, Y., Vardi, M.Y., Rozier, K.Y.: SAT-based explicit LTLf satisfiability checking. Artif. Intell. **289** (2020)
61. Lutterkort, D.: libfa. https://augeas.net/libfa/
62. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_1
63. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_14
64. Moore, E.F.: Gedanken-experiments on sequential machines. In: Automata Studies, vol. 34. Princeton University Press, Princeton (1956)
65. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

66. Muller, D., Saoudi, A., Schupp, P.: Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In: Proceedings of the LICS. IEEE (1988)
67. Møller, A., et al.: Brics automata library. https://www.brics.dk/automaton/
68. Nötzli, A., Reynolds, A., Barbosa, H., Barrett, C., Tinelli, C.: Even faster conflicts and lazier reductions for string solvers. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. CAV 2022. LNCS, vol. 13372, pp. 205–226. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13188-2_11
69. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**(6) (1987)
70. Ranzato, F., Tapparo, F.: An efficient simulation algorithm based on abstract interpretation. Inf. Comput. **208**, 1–22 (2010)
71. RegExLib.com: The Internet's first Regular Expression Library. http://regexlib.com/
72. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 149–167. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73370-6_11
73. Stanford, C., Veanes, M., Bjørner, N.S.: Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In: Proceedings of the PLDI'21. ACM (2021)
74. Tabakov, D., Vardi, M.Y.: Experimental evaluation of classical automata constructions. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 396–411. Springer, Heidelberg (2005). https://doi.org/10.1007/11591191_28
75. Valmari, A.: Simple bisimilarity minimization in O(m log n) time. Fundam. Inform. **105**(3) (2010)
76. Vardi, M.Y.: Nontraditional applications of automata theory. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 575–597. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-57887-0_116
77. Vargovčík, P., Holík, L.: Simplifying alternating automata for emptiness testing. In: Oh, H. (ed.) APLAS 2021. LNCS, vol. 13008, pp. 243–264. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-89051-3_14
78. Veanes, M.: A.NET automata library. https://github.com/AutomataDotNet/Automata
79. Veanes, M., de Halleux, P., Tillmann, N.: Rex: symbolic regular expression explorer. In: Proceedings of the ICST'10. IEEE (2010)
80. Wang, H.-E., Tsai, T.-L., Lin, C.-H., Yu, F., Jiang, J.-H.R.: String analysis via automata manipulation with logic circuit representation. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 241–260. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_13
81. Wolper, P., Boigelot, B.: An automata-theoretic approach to Presburger arithmetic constraints. In: Mycroft, A. (ed.) SAS 1995. LNCS, vol. 983, pp. 21–32. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60360-3_30
82. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: a new algorithm for checking universality of finite automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_5

Program Synthesis in Saturation

Petra Hozzová[1](✉), Laura Kovács[1], Chase Norman[2], and Andrei Voronkov[3,4]

[1] TU Wien, Vienna, Austria
petra.hozzova@tuwien.ac.at
[2] UC Berkeley, Berkeley, USA
[3] University of Manchester, Manchester, UK
[4] EasyChair, Manchester, UK

Abstract. We present an automated reasoning framework for synthesizing recursion-free programs using saturation-based theorem proving. Given a functional specification encoded as a first-order logical formula, we use a first-order theorem prover to both establish validity of this formula and discover program fragments satisfying the specification. As a result, when deriving a proof of program correctness, we also synthesize a program that is correct with respect to the given specification. We describe properties of the calculus that a saturation-based prover capable of synthesis should employ, and extend the superposition calculus in a corresponding way. We implemented our work in the first-order prover Vampire, extending the successful applicability of first-order proving to program synthesis.

Keywords: Program Synthesis · Saturation · Superposition · Theorem Proving

1 Introduction

Program synthesis constructs code from a given specification. In this work we focus on synthesis using functional specifications summarized by valid first-order formulas [1,14], ensuring that our programs are provably correct. While being a powerful alternative to formal verification [20], synthesis faces intrinsic computational challenges. One of these challenges is posed to the reasoning backend used for handling program specifications, as the latter typically include first-order quantifier alternations and interpreted theory symbols. As such, efficient reasoning with both theories and quantifiers is imperative for any effort towards program synthesis.

In this paper we address this demand for recursion-free programs. We advocate the use of first-order theorem proving for extracting code from correctness proofs of functional specifications given as first-order formulas $\forall \overline{x}.\exists y.F[\overline{x}, y]$. These formulas state that "for all (program) inputs $\overline{x}$ there exists an output y such that the input-output relation (program computation) $F[\overline{x}, y]$ is valid".

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 307–324, 2023.
https://doi.org/10.1007/978-3-031-38499-8_18

Given such a specification, we synthesize a recursion-free program while also deriving a proof certifying that the program satisfies the specification.

The programs we synthesize are built using first-order theory terms extended with `if−then−else` constructors. To ensure that our programs yield computational models, i.e., that they can be evaluated for given values of input variables $\overline{x}$, we restrict the programs we synthesize to only contain *computable* symbols.

Our Approach in a Nutshell. In order to synthesize a recursion-free program, we prove its functional specification using saturation-based theorem proving [11,15]. We extend saturation-based proof search with answer literals [5], allowing us to track substitutions into the output variable y of the specification. These substitutions correspond to the sought program fragments and are conditioned on clauses they are associated with in the proof. When we derive a clause corresponding to a program branch `if` C `then` r, where C is a condition and r a term and both C, r are computable, we store it and continue proof search assuming that $\neg C$ holds; we refer to such conditions C as (program) branch conditions. The saturation process for both proof search and code construction terminates when the conjunction of negations of the collected branch conditions becomes unsatisfiable. Then we synthesize the final program satisfying the given (and proved) specification by assembling the recorded program branches (see e.g. Examples 1–3).

The main challenges of making our approach effective come with (i) integrating the construction of the programs with `if − then − else` into the proof search, turning thus proof search into *program search/synthesis*, and (ii) guiding program synthesis to only computable branch conditions and programs.

Contributions. We bring the next contributions solving the above challenges:[1]

- We formalize the semantics for clauses with answer literals and introduce a *saturation-based algorithm for program synthesis* based on this semantics. We prove that, given a sound inference system, our saturation algorithm derives correct and computable programs (Sect. 4).
- We define properties of a sound inference calculus in order to make the calculus suitable for our saturation-based algorithm for program synthesis. We accordingly extend the superposition calculus and define a class of substitutions to be used within the extended calculus; we refer to these substitutions as *computable unifiers* (Sect. 5).
- We extend a first-order unification algorithm to find computable unifiers (Sect. 6) to be further used in saturation-based program synthesis.
- We implement our work in the Vampire prover [11] and evaluate our synthesis approach on a number of examples, complementing other techniques in the area (Sect. 7). For example, our results demonstrate the applicability of our work on synthesizing programs for specifications that cannot be even encoded in the SyGuS syntax [16].

[1] Proofs of our results are given in the extended version [8] of our paper.

2 Preliminaries

We assume familiarity with standard multi-sorted first-order logic with equality. We denote variables by x, y, terms by s, t, atoms by A, literals by L, clauses by C, D, formulas by F, G, all possibly with indices. Further, we write σ for Skolem constants. We reserve the symbol $\square$ for the *empty clause* which is logically equivalent to $\bot$. Formulas and clauses with free variables are considered implicitly universally quantified (i.e. we consider closed formulas). By $\simeq$ we denote the equality predicate and write $t \not\simeq s$ as a shorthand for $\neg t \simeq s$. We use a distinguished *integer sort*, denoted by $\mathbb{Z}$. When we use standard integer predicates $<$, $\leq$, $>$, $\geq$, functions $+, -, \ldots$ and constants $0, 1, \ldots$, we assume that they denote the corresponding interpreted integer predicates and functions with their standard interpretations. Additionally, we include a conditional term constructor if−then−else in the language, as follows: given a formula F and terms s, t of the same sort, we write $\texttt{if } F \texttt{ then } s \texttt{ else } t$ to denote the term s if F is valid and t otherwise.

An *expression* is a term, literal, clause or formula. We write $E[t]$ to denote that the expression E contains the term t. For simplicity, $E[s]$ denotes the expression E where all occurrences of t are replaced by the term s. A *substitution* θ is a mapping from variables to terms. A substitution θ is a *unifier* of two expressions E and E' if $E\theta = E'\theta$, and is a *most general unifier* (*mgu*) if for every unifier η of E and E', there exists substitution μ such that $\eta = \theta\mu$. We denote the mgu of E and E' with $\texttt{mgu}(E, E')$. We write $F_1, \ldots, F_n \vdash G_1, \ldots, G_m$ to denote that $F_1 \land \ldots \land F_n \to G_1 \lor \ldots \lor G_m$ is valid, and extend the notation also to validity modulo a theory T. Symbols occurring in a theory T are *interpreted* and all other symbols are *uninterpreted.*

2.1 Computable Symbols and Programs

We distinguish between *computable* and *uncomputable* symbols in the signature. The set of computable symbols is given as part of the specification. Intuitively, a symbol is computable if it can be evaluated and hence is allowed to occur in a synthesized program. A term or a literal is *computable* if all symbols it contains are computable. A symbol, term or literal is *uncomputable* if it is not computable.

A *functional specification*, or simply just a *specification*, is a formula

$$\forall \overline{x}. \exists y. F[\overline{x}, y]. \tag{1}$$

The variables $\overline{x}$ of a specification (1) are called *input variables.* Note that while we use specifications with a single variable y, our work can analogously be used with a tuple of variables $\overline{y}$ in (1).

Let $\overline{\sigma}$ denote a tuple of Skolem constants. Consider a computable term $r[\overline{\sigma}]$ such that the instance $F[\overline{\sigma}, r[\overline{\sigma}]]$ of (1) holds. Since $\overline{\sigma}$ are fresh Skolem constants, the formula $\forall \overline{x}. F[\overline{x}, r[\overline{x}]]$ also holds; we call such $r[\overline{x}]$ a *program* for (1) and say that the program $r[\overline{x}]$ *computes a witness* of (1).

Superposition:

$$\frac{\underline{s \simeq t} \vee C \quad \underline{L[s']} \vee C'}{(L[t] \vee C \vee C')\theta} \qquad \frac{\underline{s \simeq t} \vee C \quad \underline{u[s'] \not\simeq u'} \vee C'}{(u[t] \not\simeq u' \vee C \vee C')\theta} \qquad \frac{\underline{s \simeq t} \vee C \quad \underline{u[s'] \simeq u'} \vee C'}{(u[t] \simeq u' \vee C \vee C')\theta}$$

where $\theta := \mathtt{mgu}(s, s')$; $t\theta \not\succeq s\theta$; (first rule only) $L[s']$ is not an equality literal; and (second and third rules only) $u'\theta \not\succeq u[s']\theta$.

Binary resolution:

$$\frac{\underline{A} \vee C \quad \neg\underline{A'} \vee C'}{(C \vee C')\theta}$$

where $\theta := \mathtt{mgu}(A, A')$.

Factoring:

$$\frac{\underline{A} \vee \underline{A'} \vee C}{(A \vee C)\theta}$$

where $\theta := \mathtt{mgu}(A, A')$.

Equality resolution:

$$\frac{\underline{s \not\simeq t} \vee C}{C\theta}$$

where $\theta := \mathtt{mgu}(s, t)$.

Equality factoring:

$$\frac{\underline{s \simeq t} \vee \underline{s' \simeq t'} \vee C}{(s \simeq t \vee t \not\simeq t' \vee C)\theta}$$

where $\theta := \mathtt{mgu}(s, s')$; $t\theta \not\succeq s\theta$; and $t'\theta \not\succeq t\theta$.

Fig. 1. The superposition calculus $\mathbb{S}\mathrm{up}$.

Further, if $\forall \overline{x}.(F_1 \wedge \ldots \wedge F_n \rightarrow F[\overline{x}, r[\overline{x}]])$ holds for computable formulas $F_1, \ldots, F_n$, we write $\langle r[\overline{x}], \bigwedge_{i=1}^{n} F_i \rangle$ to refer to a *program with conditions* $F_1, \ldots, F_n$ for (1). In the sequel, we refer to (parts of) programs with conditions also as *conditional branches*. In Sect. 4 we show how to build programs for (1) by composing programs with conditions for (1) (see Corollary 3).

2.2 Saturation and Superposition

Saturation-based proof search implements *proving by refutation* [11]: to prove validity of F, a saturation algorithm establishes unsatisfiability of $\neg F$. First-order theorem provers work with clauses, rather than with arbitrary formulas. To prove a formula F, first-order provers negate F which is further skolemized and converted to clausal normal form (CNF). The CNF of $\neg F$ is denoted by $\mathtt{cnf}(\neg F)$ and represents a set S of initial clauses. First-order provers then *saturate* S by computing logical consequences of S with respect to a sound inference system $\mathcal{I}$. The saturated set of S is called the *closure* of S and the process of computing the closure of S is called *saturation*. If the closure of S contains the empty clause $\square$, the original set S of clauses is unsatisfiable, and hence the formula F is valid.

We may extend the set S of initial clauses with additional clauses $C_1, \ldots, C_n$. If C is derived by saturating this extended set, we say C is derived from S *under additional assumptions* $C_1, \ldots, C_n$.

The *superposition calculus*, denoted as $\mathbb{S}\mathrm{up}$ and given in Fig. 1, is the most common inference system used by saturation-based provers for first-order logic with equality [15]. The $\mathbb{S}\mathrm{up}$ calculus is parametrized by a *simplification ordering* $\succ$ on terms and a *selection function*, which selects in each non-empty clause a non-empty subset of literals (possibly also positive literals). We denote selected literals by underlining them. An inference rule can be applied on the given premise(s) if the literals that are underlined in the rule are also selected in the premise(s). For a certain class of selection functions, the superposition calculus

$\mathbb{S}$up is *sound* (if $\square$ is derived from F, then F is unsatisfiable) and *refutationally complete* (if F is unsatisfiable, then $\square$ can be derived from it).

2.3 Answer Literals

Answer literals [5] provide a question answering technique for tracking substitutions into given variables throughout the proof. Suppose we want to find a witness for the validity of the formula

$$\exists y.F[y]. \tag{2}$$

Within saturation-based proving, we first derive the skolemized negation of (2) and add an *answer literal* using a fresh predicate $\mathtt{ans}$ with argument y, yielding

$$\forall y.(\neg F[y] \vee \mathtt{ans}(y)). \tag{3}$$

We then saturate the CNF of (3), while ensuring that answer literals are not selected for performing inferences. If the clause $\mathtt{ans}(t_1) \vee \ldots \vee \mathtt{ans}(t_m)$ is derived during saturation, note that this clause contains only answer literals in addition to the empty clause; hence, in this case we proved unsatisfiability of $\forall y.\neg F[y]$, implying validity of (2). Moreover, $t_1, \ldots, t_m$ provides a *disjuntive answer*, i.e. witness, for the validity of (2); that is, $F[t_1] \vee \ldots \vee F[t_m]$ holds [12]. In particular, if we derive the clause $\mathtt{ans}(t)$ during saturation, we found a *definite answer* t for (2), namely $F[t]$ is valid.

Answer Literals with $\mathtt{if-then-else}$. The derivation of disjunctive answers can be avoided by modifying the inference rules to only derive clauses containing at most one answer literal. One such modification is given within the A(R)-calculus for binary resolution [22], where R is a so-called strongly liftable term restriction. The A(R)-calculus replaces the binary resolution rule when both premises contain an answer literal by the following A-resolution rule:

$$\frac{A \vee C \vee \mathtt{ans}(r) \quad \neg A' \vee C' \vee \mathtt{ans}(r')}{(C \vee C' \vee \mathtt{ans}(\mathtt{if}\ A\ \mathtt{then}\ r'\ \mathtt{else}\ r))\theta}\ (A\text{-resolution}),$$

where $\theta := \mathtt{mgu}(A, A')$ and the restriction $R(\mathtt{if}\ A\ \mathtt{then}\ r'\ \mathtt{else}\ r)$ holds.

In our work we go beyond the A-resolution rule and modify both the superposition calculus and the saturation algorithm to reason not only about answer literals but also about their use of $\mathtt{if-then-else}$ terms (see Sects. 4–5).

3 Illustrative Example

Let us illustrate our approach to program synthesis. We use answer literals in saturation to construct programs with conditions while proving specifications (1). By adding an answer literal to the skolemized negation of (1), we obtain

$$\forall y.(\neg F[\overline{\sigma}, y] \vee \mathtt{ans}(y)),$$

$$\forall x.\, i(x) * x \simeq e \;\; \text{(A1)} \qquad \forall x.\, e * x \simeq x \;\; \text{(A2)} \qquad \forall x, y, z.\, x * (y * z) \simeq (x * y) * z \;\; \text{(A3)}$$

Fig. 2. Axioms defining a group. Uninterpreted function symbols $i(\cdot), e, *$ represent the inverse, the identity element, and the group operation, respectively.

where $\overline{\sigma}$ are the skolemized input variables x. When we derive a unit clause $\texttt{ans}(r[\overline{\sigma}])$ during saturation, where $r[\overline{\sigma}]$ is a computable term, we construct a program for (1) from the definite answer $r[\overline{\sigma}]$ by replacing $\overline{\sigma}$ with the input variables $\overline{x}$, obtaining the program $r[\overline{x}]$. Hence, deriving computable definite answers by saturation allows us to synthesize programs for specifications.

Example 1. Consider the group theory axioms (A1)–(A3) of Fig. 2. We are interested in synthesizing a program for the following specification:

$$\forall x. \exists y.\ x * y \simeq e \tag{4}$$

In this example we assume that all symbols are computable. To synthesize a program for (4), we add an answer literal to the skolemized negation of (4) and convert the resulting formula to CNF (preprocessing). We consider the set S of clauses containing the obtained CNF and the axioms (A1)-(A3). We saturate S using $\mathbb{S}\text{up}$ and obtain the following derivation:[2]

1. $\sigma * y \not\simeq e \vee \texttt{ans}(y)$ [preprocessed specification]
2. $i(x) * (x * y) \simeq e * y$ [Sup A1, A3]
3. $i(x) * (x * y) \simeq y$ [Sup A2, 2.]
4. $x * y \simeq i(i(x)) * y$ [Sup 3., 3.]
5. $e \simeq x * i(x)$ [Sup 4., A1]
6. $\texttt{ans}(i(\sigma))$ [BR 5., 1.]

Using the above derivation, we construct a program for the functional specification (4) as follows: we replace σ in the definite answer $i(\sigma)$ by x, yielding the program $i(x)$. Note that for each input x, our synthesized program computes the inverse $i(x)$ of x as an output. In other words, our synthesized program for (4) ensures that each group element x has a right inverse $i(x)$.

While Example 1 yields a definite answer within saturation-based proof search, our work supports the synthesis of more complex recursion-free programs (see Examples 2–3) by composing program fragments derived in the program search (Sect. 4) as well as by using answer literals with $\texttt{if}-\texttt{then}-\texttt{else}$ to effectively handle disjunctive answers (Sect. 5).

[2] For each formula in the derivation, we also list how the formula has been derived. For example, formula 5 is the result of superposition (Sup) with formula 4 and axiom A1, whereas binary resolution (BR) has been used to derive formula 6 from 5 and 1.

4 Program Synthesis with Answer Literals

We now introduce our approach to saturation-based program synthesis using answer literals (Algorithm 1). We focus on recursion-free program synthesis and present our work in a more general setting. Namely, we consider functional specifications whose validity may depend on additional assumptions (e.g. additional program requirements) $A_1, \dots, A_n$, where each A_i is a closed formula:

$$A_1 \wedge \dots \wedge A_n \rightarrow \forall \overline{x}. \exists y. F[\overline{x}, y] \tag{5}$$

Note that specification (1) is a special case of (5). However, since $A_1, \dots, A_n$ are closed formulas, (5) is equivalent to $\forall \overline{x}. \exists y. (A_1 \wedge \dots \wedge A_n \rightarrow F[\overline{x}, y])$, which is a special case of (1).

Given a functional specification (5), we use answer literals to synthesize programs with conditions (Sect. 4.1) and extend saturation-based proof search to reason about answer literals (Sect. 4.2). For doing so, we add the answer literal $\mathtt{ans}(y)$ to the skolemized negation of (5) and obtain

$$A_1 \wedge \dots \wedge A_n \wedge \forall y. (\neg F[\overline{\sigma}, y] \vee \mathtt{ans}(y)). \tag{6}$$

We saturate the CNF of (6), while ensuring that answer literals are not selected within the inference rules used in saturation. We guide saturation-based proof search to derive clauses $C[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$, where $C[\overline{\sigma}]$ and $r[\overline{\sigma}]$ are computable.

4.1 From Answer Literals to Programs

Our next result ensures that, if we derive the clause $C[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$, the term $r[\overline{\sigma}]$ is a definite answer under the assumption $\neg C[\overline{\sigma}]$ (Theorem 1). We note that we do not terminate saturation-based program synthesis once a clause $C[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$ is derived. We rather record the program $r[\overline{x}]$ with condition $\neg C[\overline{x}]$ (and possibly also other conditions), replace clause $C[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$ by $C[\overline{\sigma}]$, and continue saturation (Corollary 2). As a result, upon establishing validity of (5), we synthesized a program for (5) (Corollary 3).

Theorem 1 [Semantics of Clauses with Answer Literals]. *Let C be a clause not containing an answer literal. Assume that, using a saturation algorithm based on a sound inference system $\mathcal{I}$, the clause $C \vee \mathtt{ans}(r[\overline{\sigma}])$ is derived from the set of clauses consisting of initial assumptions $A_1, \dots, A_n$, the clausified formula $\mathtt{cnf}(\neg F[\overline{\sigma}, y] \vee \mathtt{ans}(y))$ and additional assumptions $C_1, \dots, C_m$. Then,*

$$A_1, \dots, A_n, C_1, \dots, C_m \vdash C, F[\overline{\sigma}, r[\overline{\sigma}]].$$

That is, under the assumptions $C_1, \dots, C_m, \neg C$, the computable term $r[\overline{\sigma}]$ provides a definite answer to (5).

We further use Theorem 1 to synthesize programs with conditions for (5).

Corollary 2 [Programs with Conditions]. *Let $r[\overline{\sigma}]$ be a computable term and $C[\overline{\sigma}]$ a ground computable clause not containing an answer literal. Assume that clause $C[\overline{\sigma}]\vee\mathtt{ans}(r[\overline{\sigma}])$ is derived from the set of initial clauses $A_1,\ldots,A_n$, the clausified formula $\mathtt{cnf}(\neg F[\overline{\sigma},y]\vee\mathtt{ans}(y))$ and additional ground computable assumptions $C_1[\overline{\sigma}],\ldots,C_m[\overline{\sigma}]$, by using saturation based on a sound inference system $\mathcal{I}$. Then,*

$$\langle r[\overline{x}], \bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}]\rangle$$

is a program with conditions for (5).

Note that a program with conditions $\langle r[\overline{x}], \bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}]\rangle$ corresponds to a conditional (program) branch $\mathtt{if}\ \bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}]\ \mathtt{then}\ r[\overline{x}]$: only if the condition $\bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}]$ is valid, then $r[\overline{x}]$ is computed for (5).

We use programs with conditions $\langle r[\overline{x}], \bigwedge_{j=1}^{m} C_j[\overline{x}] \wedge \neg C[\overline{x}]\rangle$ to finally synthesize a program for (5). To this end, we use Corollary 2 to derive programs with conditions, and once their conditions cover all possible cases given the initial assumptions $A_1,\ldots,A_n$, we compose them into a program for (5).

Corollary 3 [From Programs with Conditions to Programs for (5)]. *Let $P_1[\overline{x}],\ldots,P_k[\overline{x}]$, where $P_i[\overline{x}] = \langle r_i[\overline{x}], \bigwedge_{j=1}^{i-1} C_j[\overline{x}] \wedge \neg C_i[\overline{x}]\rangle$, be programs with conditions for (5), such that $\bigwedge_{i=1}^{n} A_i \wedge \bigwedge_{i=1}^{k} C_i[\overline{x}]$ is unsatisfiable. Then $P[\overline{x}]$, given by*

$$\begin{aligned} P[\overline{x}] := {}& \mathtt{if}\ \neg C_1[\overline{x}]\ \mathtt{then}\ r_1[\overline{x}] \\ & \mathtt{else\ if}\ \neg C_2[\overline{x}]\ \mathtt{then}\ r_2[\overline{x}] \\ & \qquad \ldots \\ & \mathtt{else\ if}\ \neg C_{k-1}[\overline{x}]\ \mathtt{then}\ r_{k-1}[\overline{x}] \\ & \mathtt{else}\ r_k[\overline{x}], \end{aligned} \tag{7}$$

is a program for (5).

Note that since the conditional branches of (7) cover all possible cases to be considered over $\overline{x}$, we do not need the condition $\mathtt{if}\ \neg C_k$. In particular, if $k = 1$, i.e. $\bigwedge_{i=1}^{n} A_i \wedge C_1[\overline{x}]$ is unsatisfiable, then the synthesized program for (5) is $r_1[\overline{x}]$.

4.2 Saturation-Based Program Synthesis

Our program synthesis results from Theorem 1, Corollary 2 and Corollary 3 rely upon a saturation algorithm using a sound (but not necessarily complete) inference system $\mathcal{I}$. In this section, we present our modifications to extend state-of-the-art saturation algorithms with answer literal reasoning, allowing to derive clauses $C[\overline{\sigma}]\vee\mathtt{ans}(r[\overline{\sigma}])$, where both $C[\overline{\sigma}]$ and $r[\overline{\sigma}]$ are computable. In Sects. 5–6 we then describe modifications of the inference system $\mathcal{I}$ to implement rules over clauses with answer literals.

Algorithm 1. Saturation Loop for Recursion-Free Program Synthesis

```
1  initial set of clauses $S := \{\texttt{cnf}(A_1 \land \ldots \land A_n \land \forall y.(\neg F[\overline{\sigma}, y] \lor \texttt{ans}(y)))\}$
2  initial sets of additional assumptions $\mathcal{C} := \emptyset$ and programs $\mathcal{P} := \emptyset$
3  repeat
4     Select clause $G \in S$
5     Derive consequences $C_1, \ldots, C_n$ of $G$ and formulas from $S$ using rules of $\mathcal{I}$
6     for each $C_i$ do
7        if $C_i = (C[\overline{\sigma}] \lor \texttt{ans}(r[\overline{\sigma}]))$ and $C[\overline{\sigma}]$ is ground and computable then
8           $\mathcal{P} := \mathcal{P} \cup \{\langle r[\overline{x}], \bigwedge_{C' \in \mathcal{C}} C' \land \neg C[\overline{x}]\rangle\}$      /* Corollary 2 */
9           $\mathcal{C} := \mathcal{C} \cup \{C[\overline{x}]\}$
10          $C_i := C[\overline{\sigma}]$
11    $S := S \cup \{C_1, \ldots, C_n\}$
12    if $\square \in S$ then
13       return  program (7) for specification (5), derived from $\mathcal{P}$   /* Corollary 3 */
```

Our saturation algorithm is given in Algorithm 1. In a nutshell, we use Corollary 2 to construct programs from clauses $C[\overline{\sigma}] \lor \texttt{ans}(r[\overline{\sigma}])$ and replace clauses $C[\overline{\sigma}] \lor \texttt{ans}(r[\overline{\sigma}])$ by $C[\overline{\sigma}]$ (lines 7–10 of Algorithm 1). The newly added computable assumptions $C[\overline{\sigma}]$ are used to guide saturation towards deriving programs with conditions where the conditions contain $C[\overline{x}]$; these programs with conditions are used for synthesizing programs for (5), as given in Corollary 3.

Compared to a standard saturation algorithm used in first-order theorem proving (e.g. lines 4–5 of Algorithm 1), Algorithm 1 implements additional steps for processing newly derived clauses $C[\overline{\sigma}] \lor \texttt{ans}(r[\overline{\sigma}])$ with answer literals (lines 6–10). As a result, Algorithm 1 establishes not only the validity of the specification (5) but also synthesizes a program (lines 12–13). Throughout the algorithm, we maintain a set $\mathcal{P}$ of programs with conditions derived so far and a set $\mathcal{C}$ of additional assumptions. For each new clause C_i, we check if it is in the form $C[\overline{\sigma}] \lor \texttt{ans}(r[\overline{\sigma}])$ where $C[\overline{\sigma}]$ is ground and computable (line 7). If yes, we construct a program with conditions $\langle r[\overline{x}], \bigwedge_{C' \in \mathcal{C}} C' \land \neg C[\overline{x}]\rangle$, extend $\mathcal{C}$ with the additional assumption $C[\overline{x}]$, and replace C_i by $C[\overline{\sigma}]$ (lines 8–10). Then, when we derive the empty clause, we construct the final program as follows. We first collect all clauses that participated in the derivation of $\square$. We use this clause collection to filter the programs in $\mathcal{P}$ – we only keep a program originating from a clause $C[\overline{\sigma}] \lor \texttt{ans}(r[\overline{\sigma}])$ if the condition $C[\overline{\sigma}]$ was used in the proof, obtaining programs $P_1, \ldots, P_k$. From $P_1, \ldots, P_k$ we then synthesize the final program P using the construction (7) from Corollary 3.

Remark 1. Compared to [22] where potentially large programs (with conditions) are tracked in answer literals, Algorithm 1 removes answer literals from clauses and constructs the final program only after saturation found a refutation of the negated (5). Our approach has two advantages: first, we do not have to keep track of potentially many large terms using `if−then−else`, which might slow down saturation-based program synthesis. Second, our work can naturally be integrated with clause splitting techniques within saturation (see Sect. 7).

5 Superposition with Answer Literals

We note that our saturation-based program synthesis approach is not restricted to a specific calculus. Algorithm 1 can thus be used with *any sound* set of inference rules, including theory-specific inference rules, e.g. [10], as long as the rules allow derivation of clauses in the form $C \vee \mathtt{ans}(r)$, where C, r are computable and C is ground. I.e., the rules should only derive clauses with at most one answer literal, and should not introduce uncomputable symbols into answer literals.

In this section we present changes tailored to the superposition calculus $\mathbb{S}\mathrm{up}$, yet, without changing the underlying saturation process of Algorithm 1. We first introduce the notion of an abstract unifier [17] and define a computable unifier – a mechanism for dealing with the uncomputable symbols in the reasoning instead of introducing them into the programs. The use of such a unifier in any sound calculus is explained, with particular focus on the $\mathbb{S}\mathrm{up}$ calculus.

Definition 1 (Abstract unifier [17]). *An* abstract unifier *of two expressions E_1, E_2 is a pair (θ, D) such that:*

1. θ is a substitution and D is a (possibly empty) disjunction of disequalities,
2. $(D \vee E_1 \simeq E_2)\theta$ is valid in the underlying theory.

Intuitively speaking, an abstract unifier combines disequality constraints D with a substitution θ such that the substitution is a unifier of E_1, E_2 if the constraints D are not satisfied.

Definition 2 (Computable unifier). *A* computable unifier *of two expressions E_1, E_2 with respect to an expression E_3 is an abstract unifier (θ, D) of E_1, E_2 such that the expression $E_3\theta$ is computable.*

For example, let f be computable and g uncomputable. Then $(\{y \mapsto f(z)\}, z \not\simeq g(x))$ is a computable unifier of the terms $f(g(x)), y$ with respect to $f(y)$. Further, $(\{y \mapsto f(g(x))\}, \emptyset)$ is an abstract unifier of the same terms, but not a computable unifier with respect to $f(y)$.

Ensuring Computability of Answer Literal Arguments. We modify the rules of a sound inference system $\mathcal{I}$ to use computable unifiers with respect to the answer literal argument instead of unifiers. Since a computable unifier may entail disequality constraints D, we add D to the conclusions of the inference rules. That is, for an inference rule of $\mathcal{I}$ as below

$$\frac{C_1 \quad \cdots \quad C_n}{C\theta}\ , \tag{8}$$

where θ is a substitution such that $E\theta \simeq E'\theta$ holds for some expressions E, E', we extend $\mathcal{I}$ with the following n inference rules with computable unifiers:

$$\frac{C_1 \vee \mathtt{ans}(r) \quad C_2 \quad \cdots \quad C_n}{(\underline{D} \vee C \vee \mathtt{ans}(r))\theta'} \qquad \cdots \qquad \frac{C_1 \quad C_2 \quad \cdots \quad C_n \vee \mathtt{ans}(r)}{(\underline{D} \vee C \vee \mathtt{ans}(r))\theta'}\ , \tag{9}$$

where (θ', D) is a computable unifier of E, E' with respect to r and none of $C_1, \ldots, C_n$ contains an answer literal. We obtain the following result.

Superposition (Sup):

$$\frac{\underline{s \simeq t} \vee C \vee \mathtt{ans}(r) \quad \underline{L[s']} \vee C' \vee \mathtt{ans}(r')}{(\underline{D} \vee L[t] \vee C \vee C' \vee \mathtt{ans}(\mathtt{if}\ s \simeq t\ \mathtt{then}\ r'\ \mathtt{else}\ r))\theta} \qquad \frac{\underline{s \simeq t} \vee C \vee \mathtt{ans}(r) \quad \underline{L[s']} \vee C' \vee \mathtt{ans}(r')}{(\underline{D} \vee \underline{r \not\simeq r'} \vee L[t] \vee C \vee C' \vee \mathtt{ans}(r))\theta}$$

$$\frac{\underline{s \simeq t} \vee C \vee \mathtt{ans}(r) \quad \underline{u[s'] \not\simeq u'} \vee C' \vee \mathtt{ans}(r')}{(\underline{D} \vee u[t] \not\simeq u' \vee C \vee C' \vee \mathtt{ans}(\mathtt{if}\ s \simeq t\ \mathtt{then}\ r'\ \mathtt{else}\ r))\theta} \qquad \frac{\underline{s \simeq t} \vee C \vee \mathtt{ans}(r) \quad \underline{u[s'] \simeq u'} \vee C' \vee \mathtt{ans}(r')}{(\underline{D} \vee \underline{r \not\simeq r'} \vee u[t] \simeq u' \vee C \vee C' \vee \mathtt{ans}(r))\theta}$$

$$\frac{\underline{s \simeq t} \vee C \vee \mathtt{ans}(r) \quad \underline{u[s'] \simeq u'} \vee C' \vee \mathtt{ans}(r')}{(\underline{D} \vee u[t] \simeq u' \vee C \vee C' \vee \mathtt{ans}(\mathtt{if}\ s \simeq t\ \mathtt{then}\ r'\ \mathtt{else}\ r))\theta} \qquad \frac{\underline{s \simeq t} \vee C \vee \mathtt{ans}(r) \quad \underline{u[s'] \not\simeq u'} \vee C' \vee \mathtt{ans}(r')}{(\underline{D} \vee \underline{r \not\simeq r'} \vee u[t] \not\simeq u' \vee C \vee C' \vee \mathtt{ans}(r))\theta}$$

where (θ, D) is a computable unifier of s, s' w.r.t. the argument of the answer literal in the rule conclusion (i.e. $\mathtt{if}\ s \simeq t\ \mathtt{then}\ r'\ \mathtt{else}\ r$ for the left-column rules, and r for the others); (rules on the first line only) $L[s']$ is not an equality literal; and (rules on the second and third line only) $u'\theta \not\succeq u[s']\theta$.

Binary resolution (BR):

$$\frac{\underline{A} \vee C \vee \mathtt{ans}(r) \quad \neg\underline{A'} \vee C' \vee \mathtt{ans}(r')}{(\underline{D} \vee C \vee C' \vee \mathtt{ans}(\mathtt{if}\ A\ \mathtt{then}\ r'\ \mathtt{else}\ r))\theta} \qquad \frac{\underline{A} \vee C \vee \mathtt{ans}(r) \quad \neg\underline{A'} \vee C' \vee \mathtt{ans}(r')}{(\underline{D} \vee \underline{r \not\simeq r'} \vee C \vee C' \vee \mathtt{ans}(r))\theta}$$

where (θ, D) is a computable unifier of A, A' w.r.t. (first rule) $\mathtt{if}\ A\ \mathtt{then}\ r'\ \mathtt{else}\ r$ or (second rule) r.

Factoring (F):

$$\frac{\underline{A} \vee \underline{A'} \vee C \vee \mathtt{ans}(r)}{(\underline{D} \vee A \vee C \vee \mathtt{ans}(r))\theta}$$

where (θ, D) is a computable unifier of A, A' w.r.t. r.

Equality resolution (ER):

$$\frac{\underline{s \not\simeq t} \vee C \vee \mathtt{ans}(r)}{(\underline{D} \vee C \vee \mathtt{ans}(r))\theta}$$

where (θ, D) is a computable unifier of s, t w.r.t. r.

Equality factoring (EF):

$$\frac{\underline{s \simeq t} \vee \underline{s' \simeq t'} \vee C \vee \mathtt{ans}(r)}{(\underline{D} \vee s \simeq t \vee t \not\simeq t' \vee C \vee \mathtt{ans}(r))\theta}$$

where (θ, D) is a computable unifier of s, s' w.r.t. r; $t\theta \not\succeq s\theta$; and $t'\theta \not\succeq t\theta$.

Fig. 3. Selected rules of the extended superposition calculus $\mathbb{S}\mathrm{up}$ for reasoning with answer literals, with underlined literals being selected.

Lemma 4 [Soundness of Inferences with Answer Literals]. *If the rule* (8) *is sound, the rules* (9) *are sound as well.*

We note that we keep the original rule (8) in $\mathcal{I}$, but impose that none of its premises $C_1, \ldots, C_n$ contains an answer literal. Clearly, neither the such modified rule (8) nor the new rules (9) introduce uncomputable symbols into answer literals. Rather, these rules add disequality constraints D into their conclusions and immediately select D for further applications of inference rules. Such a selection guides the saturation process in Algorithm 1 to first discharge the constraints D containing uncomputable symbols with the aim of deriving a clause $C' \vee \mathtt{ans}(r')$ where C' is computable. The clause $C' \vee \mathtt{ans}(r')$ is then converted into a program with conditions using Corollary 2.

Superposition with Answer Literals. We make the inference rule modifications (8), together with the addition of new rules (9), for each inference rule of the $\mathbb{S}\mathrm{up}$ calculus from Fig. 1. Further, we also ensure that rules with multiple premises, when applied on several premises containing answer literals, *derive clauses with at most one answer literal.* We therefore introduce the following two rule modifications. (i) We use the $\texttt{if}-\texttt{then}-\texttt{else}$ constructor to combine answer literals of premises, by adapting the use of $\texttt{if}-\texttt{then}-\texttt{else}$ within binary resolution [13,14,22] to superposition rules. (ii) We use an answer literal from only one of the rule premises in the rule conclusion and add new disequality constraint $r \not\simeq r'$ between the premises' answer literal arguments, similar to the constraints D of the computable unifier. Analogously to the computable unifier constraints, we immediately select this disequality constraint $r \not\simeq r'$.

The resulting extension of the $\mathbb{S}$up calculus with answer literals is given in Fig. 3. In addition to the rules of Fig. 3, the extended calculus contains rules constructed as (9) for superposition and binary resolution rules of Fig. 1. Using Lemma 4, we conclude the following.

Lemma 5 [Soundness of $\mathbb{S}$up with Answer Literals]. *The inference rules from Fig. 3 of the extended $\mathbb{S}$up calculus with answer literals are sound.*

By the soundness results of Lemmas 4–5, Corollaries 2–3 imply that, when applying the calculus of Fig. 3 in the saturation-based program synthesis approach of Algorithm 1, we construct correct programs.

Example 2. We illustrate the use of Algorithm 1 with the extended $\mathbb{S}$up calculus of Fig. 3, strengthening our motivation from Sect. 3 with $\texttt{if}-\texttt{then}-\texttt{else}$ reasoning. To this end, consider the functional specification over group theory:

$$\forall x, y. \exists z. (x * y \not\simeq y * x \rightarrow z * z \not\simeq e), \tag{10}$$

asserting that, if the group is not commutative, there is an element whose square is not e. In addition to the axioms (A1)–(A3) of Fig. 2, we also use the right identity axiom (A2') $\forall x.\ x * e \simeq x$.[3] Based on Algorithm 1, we obtain the following derivation of the program for (10):

1. $\sigma_1 * \sigma_2 \not\simeq \sigma_2 * \sigma_1 \lor \texttt{ans}(z)$ [preprocessed specification]
2. $e \simeq z * z \lor \texttt{ans}(z)$ [preprocessed specification]
3. $\sigma_1 * \sigma_2 \not\simeq \sigma_2 * \sigma_1$ [answer literal removal 1. (Algorithm 1, line 10)]
4. $x * (x * y) \simeq e * y \lor \texttt{ans}(x)$ [Sup 2., A3]
5. $e \simeq x * (y * (x * y)) \lor \texttt{ans}(x * y)$ [Sup A3, 2.]
6. $x * (x * y) \simeq y \lor \texttt{ans}(x)$ [Sup 4., A2]
7. $x * e \simeq y * (x * y) \lor \texttt{ans}(\texttt{if } e \simeq x * (y * (x * y)) \texttt{ then } x \texttt{ else } x * y)$ [Sup 6., 5.]
8. $y * (x * y) \simeq x \lor \texttt{ans}(\texttt{if } e \simeq x * (y * (x * y)) \texttt{ then } x \texttt{ else } x * y)$ [Sup 7., A2']
9. $x * y \simeq y * x \lor \texttt{ans}(\texttt{if } x * (y * x) \simeq y \texttt{ then } x \texttt{ else if } e \simeq x * (y * (x * y)) \texttt{ then } x \texttt{ else } x * y)$ [Sup 6., 8.]
10. $\texttt{ans}(\texttt{if } \sigma_1 * (\sigma_2 * \sigma_1) \simeq \sigma_2 \texttt{ then } \sigma_1 \texttt{ else if } e \simeq \sigma_1 * (\sigma_2 * (\sigma_1 * \sigma_2)) \texttt{ then } \sigma_1 \texttt{ else } \sigma_1 * \sigma_2)$ [BR 9., 3.]
11. $\square$ [answer literal removal 11. (Algorithm 1, line 10)]

[3] We include axiom (A2') only to shorten the presentation of the obtained derivation.

The programs with conditions collected during saturation-based program synthesis, in particular corresponding to steps 3. and 11. above, are:

$$\begin{aligned} P_1[x,y] &:= \langle z, x*y \simeq y*x \rangle \\ P_2[x,y] &:= \langle \texttt{if } x*(y*x) \simeq y \texttt{ then } x \texttt{ else } (\texttt{if } e \simeq x*(y*(x*y)) \texttt{ then } x \texttt{ else } x*y), \\ &\qquad x*y \not\simeq y*x \rangle \end{aligned}$$

Note the variable z, representing an arbitrary witness, in $P_1[x,y]$. An arbitrary value is a correct witness in case $x{*}y \simeq y{*}x$ holds, as in this case (10) is trivially satisfied. Thus, we do not need to consider the case $x{*}y \simeq y{*}x$ separately. Hence, we construct the final program $P[x,y]$ only from $P_2[x,y]$ and obtain:

$$P[x,y] := \texttt{if } x{*}(y{*}x){\simeq}x \texttt{ then } x \texttt{ else } (\texttt{if } e{\simeq}x{*}(y{*}(x{*}y)) \texttt{ then } x \texttt{ else } x{*}y)$$

We conclude this section by illustrating the benefits of computable unifiers.

Example 3. Consider the group theory specification

$$\forall x, y. \exists z.\ z * (i(x) * i(y)) = e, \tag{11}$$

describing the inverse element z of $i(x) * i(y)$. We annotate the inverse $i(\cdot)$ as uncomputable to disallow the trivial solution $i(i(x) * i(y))$. Using computable unifiers, we synthesize the program $y * x$; that is, a program computing $y * x$ as the inverse of $i(x) * i(y)$.

6 Computable Unification with Abstraction

When compared to the $\mathbb{S}$up calculus of Fig. 1, our extended $\mathbb{S}$up calculus with answer literals from Fig. 3 uses computable unifiers instead of mgus. To find computable unifiers, we introduce Algorithm 2 by extending a standard unification algorithm [7,18] and an algorithm for unification with abstraction of [17]. Algorithm 2 combines computable unifiers with mgu computation, resulting in the computable unifier $\theta := \texttt{mgu}_{\texttt{comp}}(E_1, E_2, E_3)$ to be further used in Fig. 3.

Algorithm 2 modifies a standard unification algorithm to ensure computability of $E_3\theta$. Changes compared to a standard unification algorithm are highlighted. Algorithm 2 does not add $s \mapsto t$ to θ if s is a variable in E_3 and t is uncomputable. Instead, if t is $f(t_1, \dots, t_n)$ where f is computable but not all $t_1, \dots, t_n$ are computable, we extend θ by $s \mapsto f(x_1, \dots, x_n)$ and then add equations $x_1 = t_1, \dots, x_n = t_n$ to the set of equations $\mathcal{E}$ to be processed. Otherwise, f is uncomputable and we perform an abstraction: we consider s and t to be unified under the condition that $s \simeq t$ holds. Therefore we add a constraint $s \not\simeq t$ to the set of literals $\mathcal{D}$ which will be added to any clause invoking the computable unifier. To discharge the literal $s \not\simeq t$, one must prove $s \simeq t$. While s can be later substituted for other terms, as long as we use $\texttt{mgu}_{\texttt{comp}}$, s will never be substituted for an uncomputable term. Thus, we conclude the following result.

Theorem 6. *Let E_1, E_2, E_3 be expressions. Then $(\theta, D) := \texttt{mgu}_{\texttt{comp}}(E_1, E_2, E_3)$ is a computable unifier.*

Algorithm 2. Computable Unification with Abstraction

```
function mgu_comp($(E_1, E_2, E_3)$)
  if $E_3$ is uncomputable then fail
  let $\mathcal{E}$ be a set of equations and $\theta$ be a substitution; $\mathcal{E} := \{E_1 = E_2\}$; $\theta := \{\}$
  let $\mathcal{D}$ be a set of disequalities; $\mathcal{D} := \emptyset$
  repeat
    if $\mathcal{E}$ is empty then
      return $(\theta, D)$ where $D$ is the disjunction of literals in $\mathcal{D}$
    Select an equation $s = t$ in $\mathcal{E}$ and remove it from $\mathcal{E}$
    if $s$ coincides with $t$ then do nothing
    else if $s$ is a variable and $s$ does not occur in $t$ then
      if $s$ does not occur in $E_3$ or $t$ is computable then $\theta := \theta \circ \{s \mapsto t\}$; $\mathcal{E} = \mathcal{E}\{s \mapsto t\}$
      else if $t = f(t_1, \ldots, t_n)$ and $f$ is computable then
        $\theta := \theta \circ \{s \mapsto f(x_1, \ldots, x_n)\}$; $\mathcal{E} := \mathcal{E}\{s \mapsto f(x_1, \ldots, x_n)\} \cup \{x_1 = t_1, \ldots, x_n = t_n\}$
          where $x_1, \ldots, x_n$ are fresh variables
      else if $t = f(t_1, \ldots, t_n)$ and $f$ is uncomputable then $\mathcal{D} := \mathcal{D} \cup \{s \not\simeq t\}$
    else if $s$ is a variable and $s$ occurs in $t$ then fail
    else if $t$ is a variable then $\mathcal{E} := \mathcal{E} \cup \{t = s\}$
    else if $s$ and $t$ have different top-level symbols then fail
    else if $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ then $\mathcal{E} := \mathcal{E} \cup \{s_1 = t_1, \ldots, s_n = t_n\}$
```

7 Implementation and Experiments

Implementation. We implemented our saturation-based program synthesis approach in the Vampire prover [11]. We used Algorithm 1 with the extended $\mathbb{S}$up calculus of Fig. 3. The implementation, consisting of approximately 1100 lines of C++ code, is available at https://github.com/vprover/vampire/tree/synthesis-pr. The synthesis functionality can be turned on using the option `--question_answering synthesis`.

Vampire accepts functional specifications in an extension of the SMT-LIB2 format [4], by using the new command `assert-not` to mark the specification. We consider interpreted theory symbols to be computable. Uninterpreted symbols can be annotated as uncomputable via the command `(set-option :uncomputable (symbol1 ... symbolN))`.

Our implementation also integrates Algorithm 1 with the AVATAR architecture [26]. We modified the AVATAR framework to only allow splitting over ground computable clauses that do not contain answer literals. Further, if we derive a clause $C[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$ with AVATAR assertions $C_1[\overline{\sigma}], \ldots, C_m[\overline{\sigma}]$, where $C[\overline{\sigma}]$ is ground and computable, we replace it by the clause $C[\overline{\sigma}] \vee \bigvee_{i=1}^{m} \neg C_i[\overline{\sigma}] \vee \mathtt{ans}(r[\overline{\sigma}])$ without any assertions. We then immediately record a program with conditions $\langle r[\overline{x}], \neg C[\overline{x}] \wedge \bigwedge_{i=1}^{m} C_i[\overline{x}]\rangle$, and replace the clause by $C[\overline{\sigma}] \vee \bigvee_{i=1}^{m} \neg C_i[\overline{\sigma}]$ (see lines 7–10 of Algorithm 1), which may be then further split by AVATAR.

Finally, our implementation simplifies the programs we synthesize. If during Algorithm 1 we record a program $\langle z, F\rangle$ where z is a variable, we do not use this program in the final program construction (line 12 of Algorithm 1) even if F occurs in the derivation of $\square$ (see Example 2).

Examples and Experimental Setup. The goal of our experimental evaluation is to showcase the benefits of our approach on problems that are deemed to be hard, even unsolvable, by state-of-the-art synthesis techniques. We therefore focused on first–order theory reasoning and evaluated our work on the group theory problems of Examples 1–3, as well as on integer arithmetic problems.

As the SMT-LIB2 format can easily be translated into the SyGuS 2.1 syntax [16], we compared our results to cvc5 1.0.4 [3], supporting SyGuS-based synthesis [2]. Our experiments were run on an AMD Epyc 7502, 2.5 GHz CPU with 1 TB RAM, using a 5 min time limit per example. Our benchmarks as well as the configurations for our experiments are available at: https://github.com/vprover/vampire_benchmarks/tree/master/synthesis

Experimental Results with Group Theory Properties. Vampire synthesizes the solutions of the Examples 1–3 in 0.01, 13, and 0.03 s, respectively. Since these examples use uninterpreted functions, they cannot be encoded in the SyGuS 2.1 syntax, showcasing the limits of other synthesis tools.

Experimental Results with Maximum of $n \geq 2$ Integers. For the maximum of 2 integers, the specification is $\forall x_1, x_2 \in \mathbb{Z}.\ \exists y \in \mathbb{Z}.\big(y \geq x_1 \wedge y \geq x_2 \wedge (y = x_1 \vee y = x_2)\big)$, and the program we synthesize is `if` $x_1 < x_2$ `then` x_2 `else` x_1. Both our work and cvc5 are able to synthesize programs choosing the maximal value for up to $n = 23$ input variables, as summarized below. For $n > 23$, both Vampire and cvc5 time out.

Number n of variables for which max is synthesized	2	5	10	15	20	22	23
Vampire	0.03	0.03	0.05	1	13	55	215
cvc5	0.01	0.03	0.6	6.8	88	188	257

Experimental Results with Polynomial Equations. Vampire can synthesize the solution of polynomial equations; for example, for $\forall x_1, x_2 \in \mathbb{Z}. \exists y \in \mathbb{Z}.(y^2 = x_1^2 + 2x_1x_2 + x_2^2)$, we synthesize $x_1 + x_2$. Vampire finds the corresponding program in 26 s using simple first-order reasoning, while cvc5 fails in our setup.

8 Related Work

Our work builds upon deductive synthesis [14] adapted for the resolution calculus [13,22]. We extend this line of work with saturation-based program synthesis, by using adjustments of the superposition calculus.

Component-based synthesis of recursion-free programs [21] from logical specifications is addressed in [6,21,24]. The work of [21] uses first-order theorem proving to prove specifications and extract programs from proofs. In [6,24], ∃∀ formulas are produced to capture specifications over component properties and

SMT solving is applied to find a term satisfying the formula, corresponding to a straight-line program. We complement [21] with saturation-based superposition proving and avoid template-based SMT solving from [6,24].

A prominent line of research comes with syntax guided synthesis (SyGuS) [1], where functional specifications are given using a context-free grammar. This grammar yields program templates to be synthesized via an enumerative search procedure based on SMT solving [3,9]. We believe our work is complementary to SyGuS, by strengthening first-order reasoning for program synthesis, as evidenced by Examples 1–3.

The sketching technique [19,25] synthesizes program assignments to variables, using an alternative framework to the program synthesis setting we rely upon. In particular, sketching addresses domains that do not involve input logical formulas as functional specifications, such as example-guided synthesis [23].

9 Conclusions

We extend saturation-based proof search to saturation-based program synthesis, aiming to derive recursion-free programs from specifications. We integrate answer literals with saturation, and modify the superposition calculus and unification to synthesize computable programs. Our initial experiments show that a first-order theorem prover becomes an efficient program synthesizer, potentially opening up interesting avenues toward recursive program synthesis, for example using saturation-based proving with induction.

Acknowledgements. We thank Haniel Barbosa for support with experiments with cvc5. This work was partially funded by the ERC CoG ARTIST 101002685, and the FWF grants LogiCS W1255-N23 and LOCOTES P 35787.

References

1. Alur, R., et al.: Syntax-guided synthesis. In: Dependable Software Systems Engineering, pp. 1–25. IOS Press (2015)
2. Alur, R., Fisman, D., Padhi, S., Reynolds, A., Singh, R., Udupa, A.: SyGuS-Comp 2019 (2019). https://sygus.org/comp/2019/
3. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard Version 2.6 (2021). https://www.SMT-LIB.org
5. Green, C.: Theorem-proving by resolution as a basis for question-answering systems. Mach. Intell. **4**, 183–205 (1969)
6. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI, pp. 62–73 (2011)
7. Hoder, K., Voronkov, A.: Comparing unification algorithms in first-order theorem proving. In: Mertsching, B., Hund, M., Aziz, Z. (eds.) KI 2009. LNCS (LNAI), vol. 5803, pp. 435–443. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04617-9_55

8. Hozzová, P., Kovács, L., Norman, C., Voronkov, A.: Program synthesis in saturation. EasyChair Preprint no. 10223 (EasyChair, 2023). https://easychair.org/publications/preprint/KRmQ
9. Jha, S., Seshia, S.A.: A theory of formal synthesis via inductive learning. Acta Informatica **54**(7), 693–726 (2017)
10. Korovin, K., Kovács, L., Reger, G., Schoisswohl, J., Voronkov, A.: ALASCA: reasoning in quantified linear arithmetic. In: Sankaranarayanan, S., Sharygina, N. (eds.) TACAS 2023. LNCS, vol. 13993, pp. 647–665. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30823-9_33
11. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
12. Kunen, K.: The semantics of answer literals. J. Autom. Reason. **17**(1), 83–95 (1996)
13. Lee, R.C.T., Waldinger, R.J., Chang, C.L.: An improved program-synthesizing algorithm and its correctness. Commun. ACM **17**(4), 211–217 (1974)
14. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**(1), 90–121 (1980)
15. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Handbook of Automated Reasonings, vol. I, pp. 371–443. Elsevier and MIT Press (2001)
16. Padhi, S., Polgreen, E., Raghothaman, M., Reynolds, A., Udupa, A.: The SyGuS Language Standard Version 2.1 (2021). https://sygus.org/language/
17. Reger, G., Suda, M., Voronkov, A.: Unification with abstraction and theory instantiation in saturation-based reasoning. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 3–22. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_1
18. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965)
19. Solar-Lezama, A.: The sketching approach to program synthesis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 4–13. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10672-9_3
20. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: POPL, pp. 313–326 (2010)
21. Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., Underwood, I.: Deductive composition of astronomical software from subroutine libraries. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 341–355. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58156-1_24
22. Tammet, T.: Completeness of resolution for definite answers. J. Logic Comput. **5**(4), 449–471 (1995)
23. Thakkar, A., Naik, A., Sands, N., Alur, R., Naik, M., Raghothaman, M.: Example-guided synthesis of relational queries. In: PLDI, pp. 1110–1125 (2021)
24. Tiwari, A., Gascón, A., Dutertre, B.: Program synthesis using dual interpretation. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 482–497. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_33
25. Torlak, E., Bodik, R.: Growing solver-aided languages with rosette. In: Onward!, pp. 135–152. Onward! 2013 (2013)
26. Voronkov, A.: AVATAR: the architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 696–710. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_46

A Uniform Formalisation of Three-Valued Logics in Bisequent Calculus

Andrzej Indrzejczak(✉) and Yaroslav Petrukhin

Department of Logic, University of Lodz, Łódź, Poland
andrzej.indrzejczak@filhist.uni.lodz.pl,
iaroslav.petrukhin@edu.uni.lodz.pl

Abstract. We present a uniform characterisation of three-valued logics by means of bisequent calculus (BSC). It is a generalised form of sequent calculus (SC) where rules operate on the ordered pairs of ordinary sequents. BSC may be treated as the weakest kind of system in the rich family of generalised SC operating on items being some collections of ordinary sequents. This family covers several forms of hypersequent and nested sequent calculi introduced to provide decent SC for several non-classical logics. It seems that for many non-classical logics, including some many-valued, paraconsistent and modal logics, this reasonably modest generalization of standard SC is sufficient. In this paper we examine a variety of three-valued logics and show how they can be formalised in the framework of bisequent calculus. All provided systems are cut-free and satisfy the subformula property. Also the interpolation theorem is constructively proved for some logics.

Keywords: Bisequent Calculus · Cut elimination · Many-valued Logic · Three-valued logic · Interpolation Theorem

1 Introduction

The aim of this paper is to provide a uniform characterization of a variety of three-valued logics by means of a simple cut-free generalised sequent calculus (SC) called bisequent calculus (BSC). It is the weakest kind of system in the rich family of generalised sequent calculi operating on collections of ordinary sequents [23]. If we restrict our interest to structures built of two sequents only, we obtain a limiting case of either hypersequent or nested sequent calculi; it is what we call bisequent calculus.

Is such restricted calculus of any use? Hypersequent calculi already may be seen as a quite restrictive form of generalised SC, yet they were shown to be useful in many fields (see, e.g., [25] for a survey of applications of hypersequent calculi

Funded by the European Union (ERC, ExtenDD, project number: 101054714). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 325–343, 2023.
https://doi.org/10.1007/978-3-031-38499-8_19

in modal logic, and [37] for their use in fuzzy logic). BSC is even more restrictive but preliminary work on its application is promising. It was already successfully applied to first-order modal logic **S5** [23] and to the class of four-valued quasi-relevant logics [27]. In what follows we will focus on another application of such minimal framework – to three-valued logics.

Several proof systems of different kinds were proposed so far for many-valued logics (see e.g. Hähnle [20] for a survey). The most direct and popular approach to construction of many-valued sequent or tableau systems is based on the idea of syntactic representation of n values either by means of n-sided sequents (e.g. [8,45,56]) or by n labels attached to formulae or sets of formulae (e.g. [11,53,55]). This solution was presented by many authors and despite its popularity has many drawbacks (see [25] for discussion). Significant improvement in the construction of efficient SC or tableau systems for many-valued logic was proposed independently by Doherty [15] and Hähnle [19], where labels correspond not to single values but to their sets (sets-as signs). Among other proof-theoretic approaches to many-valued logics let us mention Caleiro and Marcelino's [10] analytic calculi for many-valued non-deterministic logics as well as the result by Grätz [18] who has recently developed analytic tableau systems based on sets-as-signs DNF representations with a correspondence to canonical sequent calculi.

Although BSC is a strictly syntactical calculus its semantical interpretation makes it similar to set-as-signs approach. A fuller discussion of this issue is provided in [27]. BSC is uniform in the sense that all three-valued logics are characterised by the same set of axiomatic sequents, and in the case of logics having the same set of connectives (i.e. defined in the same way) the rules are identical even if the set of designated values or the consequence relation is defined in different way. In this sense BSC is more uniform than several other approaches where either the set of axioms must be changed or rules for connectives must be different (even if described by means of the same table). In particular, BSC is superior in this respect to the generalised calculus presented in [25].

Section 2 has rather encyclopaedic character and provides self-contained description of a representative selection of three-valued logics. Section 3 contains a case study of BSC for $\mathbf{K}_3$ and **LP**. In Sect. 4 we provide rules for connectives of all logics introduced in Sect. 2. Section 5 shows how BSC can be applied to prove interpolation for some three-valued paraconsistent and paracomplete logics. We finish with remarks on possible extensions and comparison with other approaches to formalisation of many-valued logics.

2 Logics

We will examine several three-valued propositional logics determined by three element matrices with classical-like connectives (negation, disjunction, conjunction, and implication, plus the usual three-valued modal-style connectives); we are not going to consider other types of connectives because of the lack of space. The languages of these logics are freely generated algebras similar to three element algebras of values. Logics are interpreted by homomorphisms from languages to algebras such that $h(c^n(\varphi_1, \ldots, \varphi_n)) = \underline{c}(h(\varphi_1), \ldots, h(\varphi_n))$ for every $n-$ary connective c and the corresponding operation $\underline{c}$.

Let us consider as the starting point two three element Kleene's algebras of the form: $\mathfrak{A}_3 = \langle A, O \rangle$ where $A = \{0, u, 1\}$ and O contains an unary operation $\underline{\neg} : A \longrightarrow A$ and binary operations $\odot : A \times A \longrightarrow A$, where $\odot \in \{\underline{\wedge}, \underline{\vee}, \underline{\rightarrow}\}$. The operations are defined by the following truth tables in the strong and weak Kleene algebra; the latter considered also by Bochvar [9] (negation is the same in both):

$\underline{\wedge}$	1	u	0
1	1	u	0
u	u	u	0
0	0	0	0

$\underline{\vee}$	1	u	0
1	1	1	1
u	1	u	u
0	1	u	0

$\underline{\rightarrow}$	1	u	0
1	1	u	0
u	1	u	u
0	1	1	1

	$\underline{\neg}$
1	0
u	u
0	1

$\underline{\wedge_w}$	1	u	0
1	1	u	0
u	u	u	u
0	0	u	0

$\underline{\vee_w}$	1	u	0
1	1	u	1
u	u	u	u
0	1	u	0

$\underline{\rightarrow_w}$	1	u	0
1	1	u	0
u	u	u	u
0	1	u	1

We obtain four matrices by specifying a set of designated values D either as $\{1\}$ or $\{1, u\}$. These are called $\mathfrak{S}\mathfrak{M}^3_1$, $\mathfrak{S}\mathfrak{M}^3_2$, $\mathfrak{W}\mathfrak{M}^3_1$ and $\mathfrak{W}\mathfrak{M}^3_2$ (where $\mathfrak{S}$ stands for strong, $\mathfrak{W}$ for weak, 1 and 2 indicate the amount of designated values). In general we will call matrices with $D = \{1\}$ 1-matrices, and with $D = \{1, u\}$ – 2-matrices. Accordingly we will also call logics determined by 1-matrices and 2-matrices, 1- and 2-logics respectively. For any matrix we define a relation of matrix consequence in the standard way:

$$\varGamma \models_M \varphi \text{ iff for any homomorphism } h: \text{ if } h(\varGamma) \subseteq D, \text{ then } h(\varphi) \in D.$$

Logics are identified with their matrix consequences. In particular, logics determined by these matrices are $\mathbf{K}_3$ (strong Kleene 1-logic) [31], **LP** – the logic of paradox of Asenjo and Priest (corresponding 2-logic) [2,42], $\mathbf{K}^{\mathbf{w}}_3$ (weak Kleene 1-logic) [31], **PWK** (paraconsistent weak Kleene 2-logic) of Halldén [21].

Let us consider a few modifications of strong and weak Kleene logics. Here is McCarthy's logic $\mathbf{K}^{\rightarrow}_3$ [36] (also called Kleene's sequential and studied by Fitting [16]) and its interesting modification presented by Komendantskaya [32] under the name $\mathbf{K}^{\leftarrow}_3$ by means of the following truth tables (again, negation is unchanged):

$\underline{\wedge_{mC}}$	1	u	0
1	1	u	0
u	u	u	u
0	0	0	0

$\underline{\vee_{mC}}$	1	u	0
1	1	1	1
u	u	u	u
0	1	u	0

$\underline{\rightarrow_{mC}}$	1	u	0
1	1	u	0
u	u	u	u
0	1	1	1

$\underline{\wedge_K}$	1	u	0
1	1	u	0
u	u	u	0
0	0	u	0

$\underline{\vee_K}$	1	u	0
1	1	u	1
u	1	u	u
0	1	u	0

$\underline{\rightarrow_K}$	1	u	0
1	1	u	0
u	1	u	u
0	1	u	1

Both $\mathbf{K}^{\rightarrow}_3$ and $\mathbf{K}^{\leftarrow}_3$ are logics determined by 1-matrices. An important property of $\mathbf{K}_3$, $\mathbf{K}^{\mathbf{w}}_3$, $\mathbf{K}^{\rightarrow}_3$, and $\mathbf{K}^{\leftarrow}_3$ is that they are the only three-valued logics with one designated value which produce partial recursive predicates (see [31,32] for more details).

Several other important logics are obtained by changing the definitions of → and ¬. Consider Łukasiewicz's [34], Słupecki's [49], Heyting's [22] implications as well as Heyting's [22], Bochvar's [9], Post's and dual Post's [41] negations. Let us also consider yet another pair of additive conjunction and disjunction, arising in Łukasiewicz's logic:

$\to_L$	1	u	0
1	1	u	0
u	1	1	u
0	1	1	1

$\to_{Sl}$	1	u	0
1	1	u	0
u	1	1	1
0	1	1	1

$\to_H$	1	u	0
1	1	u	0
u	1	1	0
0	1	1	1

	$\neg_H$
1	0
u	0
0	1

	$\neg_B$
1	0
u	1
0	1

	$\neg_P$
1	u
u	0
0	1

	$\neg_{DP}$
1	0
u	1
0	u

$\wedge_L$	1	u	0
1	1	u	0
u	u	0	0
0	0	0	0

$\vee_L$	1	u	0
1	1	1	1
u	1	1	u
0	1	u	0

$\mathfrak{SM}_1^3$ with Łukasiewicz's implication (instead of Kleene's one) yields famous Łukasiewicz's $\mathbf{Ł}_3$, the first many-valued logic. In $\mathbf{Ł}_3$ we may deal with two pairs of conjunction and disjunction. We have: $\varphi \vee \psi = (\varphi \to_L \psi) \to_L \psi$ and $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, but $\varphi \wedge_L \psi = \neg(\varphi \to_L \neg\psi)$ and $\varphi \vee_L \psi = \neg\varphi \to_L \psi$. $\mathfrak{SM}_1^3$ with Słupecki's implication is an alternative to $\mathbf{Ł}_3$ having the deduction theorem. It was studied by Słupecki, Bryll, and Prucnal [49] as well as Avron [4], under the name $\mathbf{GM}_3$. If we change negation and implication of $\mathfrak{SM}_1^3$ to Heyting's ones, then we get Heyting's [22] logic $\mathbf{G}_3$, a close relative of intuitionistic logic (the name after Gödel who also studied it [17]; this logic was investigated by Jaśkowski as well [29]). The disjunction of $\mathfrak{SM}_1^3$ and Post's cyclic negation from Post's logic $\mathbf{P}_3$ [41] which is known for being functionally complete in the three-valued setting. In [40], a dual cyclic negation $\neg_{DP}$ was suggested (it reverses the direction of cyclicality of Post's negation). $\mathfrak{SM}_2^3$ with Heyting's implication and Bochvar's negation was investigated by Osorio and Carballido [38] under the name $\mathbf{G}'_3$. In the case of $\mathfrak{SM}_2^3$ the following connectives are interesting as well: Sobociński's [51] conjunction, disjunction, and implications as well as D'Ottaviano/DaCosta/Jaśkowski/Słupecki's implication [13,30,48]:

$\wedge_S$	1	u	0
1	1	1	0
u	1	u	0
0	0	0	0

$\vee_S$	1	u	0
1	1	1	1
u	1	u	0
0	1	0	0

$\to_S$	1	u	0
1	1	0	0
u	1	u	0
0	1	1	1

$\to'_S$	1	u	0
1	1	u	0
u	1	1	0
0	1	u	1

$\to_J$	1	u	0
1	1	u	0
u	1	u	0
0	1	1	1

Sobociński's logic $\mathbf{S}_3$ is obtained from $\mathfrak{SM}_2^3$ by the replacement of all binary connectives of this matrix with Sobociński's original ones. This logic may be treated as a relevant logic. However, a more popular three-valued relevant logic is Anderson and Belnap's $\mathbf{RM}_3$ [1] which is obtained from $\mathfrak{SM}_2^3$ only by the replacement of its implication with Sobociński's one. Note that earlier Sobociński [50] considered yet another implication $\to'_S$. $\mathfrak{SM}_2^3$ with the implication due to D'Ottaviano/DaCosta/Jaśkowski/Słupecki (first mentioned by Słupecki [48]) instead of Kleene's one was independently studied by several authors: D'Ottaviano and da Costa themselves [13,14], Asenjo and Tamburino [3], Batens [7] (under the name PI^s), Avron [5] (under the name $\mathbf{RM}_3^{\supset}$), and Rozonoer [44] (under the name **PCont**). An important extension of this logic is $\mathbf{J}_3$ by D'Ottaviano and da Costa [13]. It has an additional connective which is Łukasiewicz's tabular possibility operator (see below; we also present Łukasiewicz's tabular necessity operator).

$\wedge_C$	1	u	0
1	1	0	0
u	0	0	0
0	0	0	0

$\vee_C$	1	u	0
1	1	1	1
u	1	0	0
0	1	0	0

$\to_C$	1	u	0
1	1	0	0
u	1	1	1
0	1	1	1

	$\Diamond$
1	1
u	1
0	0

	$\Box$
1	1
u	0
0	0

As it is easy to guess, since $\mathbf{RM}_3$ may be viewed as a relevant logic, it should be paraconsistent as well. Moreover, $\mathbf{J}_3$, $\mathbf{S}_3$, **LP**, and many other three-valued logics with two designated values are paraconsistent (in contrast, three-valued logics with one designated value are paracomplete). One of the most famous three-valued paraconsistent logics is Sette's logic $\mathbf{P}^1$ [47]. It has Bochvar's negation and the above presented binary connectives (both 1 and u are designated). There is a version of $\mathbf{P}^1$ with Kleene's negation introduced by Carnielli and Marcos [12,35] and called $\mathbf{P}^2$. A paracomplete companion of $\mathbf{P}^1$, the logic $\mathbf{I}^1$, was presented by Sette and Carnielli [46]: it has Heyting's negation and presented below binary connectives (the implication has been first introduced by Bochvar [9]). Its version with Kleene's negation is $\mathbf{I}^2$ due to Marcos [35]. Both $\mathbf{I}^1$ and $\mathbf{I}^2$ have one designated value.

$\wedge_{Se}$	1	u	0
1	1	1	0
u	1	1	0
0	0	0	0

$\vee_{Se}$	1	u	0
1	1	1	1
u	1	1	1
0	1	1	0

$\rightarrow_{Se}$	1	u	0
1	1	1	0
u	1	1	0
0	1	1	1

$\rightarrow_{R}$	1	u	0
1	1	0	0
u	1	1	0
0	1	1	1

$\rightarrow_{T}$	1	u	0
1	1	0	0
u	1	1	u
0	1	1	1

Last but not least, let us mention Rescher's [43] and Tomova's [57] implications (added above). These implications can be added to $\mathfrak{SM}_1^3$. Tomova [57] introduced the concept of natural implication. In three-valued case with one designated value there are only 6 natural implications: Łukasiewicz's, Słupecki's, Heyting's, Bochvar's, Rescher's, and Tomova's. In the case with two designated values there are 24 natural implications, including Heyting's and Rescher's as well as D'Ottaviano/DaCosta/Jaśkowski/Słupecki's implication, both Sobociński's implications, and Sette's implication.

3 Bisequent Calculus for $\mathbf{K}_3$ (and LP)

Bisequents in BSC are ordered pairs of sequents $\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma$, where $\Gamma, \Delta, \Pi, \Sigma$ are finite (possibly empty) multisets of formulae. We will call the left component of a bisequent as 1-sequent and the right as 2-sequent respectively. Bisequents with all elements being atomic will be also called atomic. In what follows B stands for arbitrary bisequents and S for sequents.

Let us define the calculus BSC-K$_3$ which provides an adequate formalisation of $\mathbf{K}_3$. A bisequent $\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma$ is axiomatic iff it has nonempty $\Gamma \cap \Sigma$ or $\Gamma \cap \Delta$ or $\Pi \cap \Sigma$. In fact this set of axioms is fixed for all considered calculi. If constants $\top, \bot, U$ (the last for fixed undefined proposition) are added we must add axioms of the form: $\Gamma \Rightarrow \Delta, \top \mid \Pi \Rightarrow \Sigma$; $\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \top$; $\bot, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma$; $\Gamma \Rightarrow \Delta \mid \bot, \Pi \Rightarrow \Sigma$; $U, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma$ and $\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, U$.

The set of rules characterising the operations of the strong Kleene algebra consists of the following schemata:

$$(\neg{\Rightarrow}\mid)\ \frac{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi}{\neg\varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma} \quad ({\Rightarrow}\neg\mid)\ \frac{\Gamma \Rightarrow \Delta \mid \varphi, \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta, \neg\varphi \mid \Pi \Rightarrow \Sigma}$$

$$(\mid\neg{\Rightarrow})\ \frac{\Gamma \Rightarrow \Delta, \varphi \mid \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \neg\varphi, \Pi \Rightarrow \Sigma} \quad (\mid{\Rightarrow}\neg)\ \frac{\varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \neg\varphi}$$

$$(\wedge\Rightarrow|)\ \frac{\varphi, \psi, \Gamma \Rightarrow \Delta \mid S}{\varphi \wedge \psi, \Gamma \Rightarrow \Delta \mid S} \quad (\Rightarrow\wedge\ |)\ \frac{\Gamma \Rightarrow \Delta, \varphi \mid S \qquad \Gamma \Rightarrow \Delta, \psi \mid S}{\Gamma \Rightarrow \Delta, \varphi \wedge \psi \mid S}$$

$$(|\ \wedge\Rightarrow)\ \frac{S \mid \varphi, \psi, \Gamma \Rightarrow \Delta}{S \mid \varphi \wedge \psi, \Gamma \Rightarrow \Delta} \quad (|\Rightarrow\wedge)\ \frac{S \mid \Gamma \Rightarrow \Delta, \varphi \qquad S \mid \Gamma \Rightarrow \Delta, \psi}{S \mid \Gamma \Rightarrow \Delta, \varphi \wedge \psi}$$

$$(\Rightarrow\vee\ |)\ \frac{\Gamma \Rightarrow \Delta, \varphi, \psi \mid S}{\Gamma \Rightarrow \Delta, \varphi \vee \psi \mid S} \quad (\vee\Rightarrow\ |)\ \frac{\varphi, \Gamma \Rightarrow \Delta \mid S \qquad \psi, \Gamma \Rightarrow \Delta \mid S}{\varphi \vee \psi, \Gamma \Rightarrow \Delta \mid S}$$

$$(|\Rightarrow\vee)\ \frac{S \mid \Gamma \Rightarrow \Delta, \varphi, \psi}{S \mid \Gamma \Rightarrow \Delta, \varphi \vee \psi} \quad (|\ \vee\Rightarrow)\ \frac{S \mid \varphi, \Gamma \Rightarrow \Delta \qquad S \mid \psi, \Gamma \Rightarrow \Delta}{S \mid \varphi \vee \psi, \Gamma \Rightarrow \Delta}$$

$$(\Rightarrow\rightarrow\ |)\ \frac{\Gamma \Rightarrow \Delta, \psi \mid \varphi, \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta, \varphi \rightarrow \psi \mid \Pi \Rightarrow \Sigma} \qquad (|\Rightarrow\rightarrow)\ \frac{\varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \psi}{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi \rightarrow \psi}$$

$$(\rightarrow\Rightarrow\ |)\ \frac{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi \quad \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}{\varphi \rightarrow \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}$$

$$(|\rightarrow\Rightarrow)\ \frac{\Gamma \Rightarrow \Delta, \varphi \mid \Pi \Rightarrow \Sigma \quad \Gamma \Rightarrow \Delta \mid \psi, \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \varphi \rightarrow \psi, \Pi \Rightarrow \Sigma}$$

Note that all rules satisfy the subformula property and other desirable properties of well-behaved SC. In particular, they are context independent in the sense that validity-preservation of rules is intact by deletion or addition of the same parameters in the premisses and conclusion. This feature will be of special importance for the proof of the interpolation theorem. One may easily observe that in case of the rules for strong $\wedge, \vee$ we have just standard G3 rules but repeated in both components. Rules for negation and implication have different character since side and principal formula are in different sequents in all cases.

Bisequents as such do not directly correspond to standard consequence relations in suitable matrices. Hence before we define the notion of a proof in BSC-K$_3$ (or any other logic) it is better to start with more general concept. A proof-search tree for a bisequent B in BSC-L, where L is any logic, is a tree of bisequents with B as the root and nodes generated by rules of BSC-L. A proof-search tree is complete iff every leaf is atomic, and it is axiomatic iff all leaves are axiomatic. The height of a proof-search tree is defined as the length of the maximal branches. A simple consequence of the subformula property of rules is:

Proposition 1. *Every proof-search tree may be extended to a complete proof-search tree.*

The notion of a proof in BSC-K$_3$ is introduced not only by restricting the class of proof-search trees in BSC-K$_3$ to axiomatic ones but also by restricting the class of admissible roots. In general the rationale for bisequents is that 1-sequent corresponds to consequence relation in 1-matrices and 2-sequent to consequence relation in 2-matrices. Since $\mathbf{K}_3$ is characterised by 1-matrix we have:

BSC-K$_3 \vdash B$ iff there is an axiomatic proof-search tree for $B := \Gamma \Rightarrow \varphi \mid\Rightarrow$.

We define the L-validity (L-satisfiability) of bisequents in the following way:

$\mathbf{L} \models \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma$ iff every homomorphism h satisfies $\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma$. The latter holds for h iff for some φ: either ($\varphi \in \Gamma$ and $h(\varphi) \neq 1$) or ($\varphi \in \Delta$ and $h(\varphi) = 1$) or ($\varphi \in \Pi$ and $h(\varphi) = 0$) or ($\varphi \in \Sigma$ and $h(\varphi) \neq 0$).

Clearly $\mathbf{L} \not\models \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma$ iff for some h, all elements of Γ are true, all elements of Δ are either false or undefined, all elements of Π are either true or undefined and all elements of Σ are false. In this case we say that h falsifies this sequent.

Obviously, all axiomatic bisequents are valid for any logic L. As for the rules they are not only sound (i.e. validity-preserving) but also invertible; namely it holds:

Theorem 1. *For all rules of BSC-K*$_3$*, all premisses are* $\mathbf{K}_3$*-valid iff the conclusion is* $\mathbf{K}_3$*-valid.*

Proof. Straightforward proof by tedious checking. □

A simple consequence of this theorem is that for every rule the conclusion is falsified by some h iff at least one premiss is falsified by the same h.

Theorem 2 (Soundness). *If BSC-K*$_3 \vdash \Gamma \Rightarrow \varphi \mid \Rightarrow$*, then* $\Gamma \models_{K_3} \varphi$

Proof. By induction on the height of the proof, use Theorem 1. □

Invertibility of all rules implies that proof search process is confluent, i.e. that the order of applications of rules does not affect the result. In particular, B is provable iff every proof-search tree may be extended to obtain a proof.

Theorem 3 (Completeness). *If* $\Gamma \models_{K_3} \varphi$*, then BSC-K*$_3 \vdash \Gamma \Rightarrow \varphi \mid \Rightarrow$*.*

Proof. Assume that $\Gamma \models_{K_3} \varphi$ but BSC-K$_3 \nvdash \Gamma \Rightarrow \varphi \mid \Rightarrow$. Hence in every complete proof-search tree for $\Gamma \Rightarrow \varphi \mid \Rightarrow$ there is at least one branch starting with non-axiomatic atomic bisequent falsified by some h. Since all rules inherit this valuation, then the root is also falsified contrary to our assumption. □

As a simple consequence we obtain also a decision procedure for $\mathbf{K}_3$ (and for other logics L with complete BSC-L). Another by-product of our proof is that the following cut rules are admissible in BSC-K$_3$ (and other logics):

$$(Cut\mid)\quad \frac{\Gamma \Rightarrow \Delta, \varphi \mid \Lambda \Rightarrow \Theta \qquad \varphi, \Pi \Rightarrow \Sigma \mid \Xi \Rightarrow \Omega}{\Gamma, \Pi \Rightarrow \Delta, \Sigma \mid \Lambda, \Xi \Rightarrow \Theta, \Omega}$$

$$(\mid Cut)\quad \frac{\Gamma \Rightarrow \Delta \mid \Lambda \Rightarrow \Theta, \varphi \qquad \Pi \Rightarrow \Sigma \mid \varphi, \Xi \Rightarrow \Omega}{\Gamma, \Pi \Rightarrow \Delta, \Sigma \mid \Lambda, \Xi \Rightarrow \Theta, \Omega}$$

Moreover, we can constructively prove that these cut rules are admissible in the same way as it is done for four-valued logics in [27]. Due to lack of space we omit this issue here.

Note that the rules stated above provide BSC not only for $\mathbf{K}_3$ but also for **LP**. The only difference is that in **LP** we consider as provable all bisequents of the form $\Rightarrow \mid \Gamma \Rightarrow \varphi$, which is a consequence of the fact that it is determined by 2-matrix. All the results established for BSC-K$_3$ hold for BSC-LP.

4 Bisequent Calculi for Other Logics

We provide sets of rules adequate for all logics described in Sect. 2. Every operation will be characterised by four rules of introduction to antecedents and consequents of 1- and 2-sequent. The rules are devised on the basis of geometrical insights based on the tabular representation of the respective connective: to establish the premisses for the rule with the principal formula in one of the four positions in a bisequent, we just examine its tabular representation. For example, if indicated values of the arguments form a rectangle, one premiss is enough, in case of more complex shapes, two or three premisses are required. Since the process of construction of rules on the basis of tables is not deterministic we do not propose any algorithm for that aim, however by the end of this section we will illustrated the method with one example. In every case it holds that either:

$$\Gamma \models_L \varphi \text{ iff BSC-L} \vdash \Gamma \Rightarrow \varphi \mid \Rightarrow \quad \text{or} \quad \Gamma \models_L \varphi \text{ iff BSC-L} \vdash {\Rightarrow} \mid \Gamma \Rightarrow \varphi$$

depending on the fact whether $\models_L$ denotes consequence relation for logics characterised by 1-matrices or by 2-matrices. Adequacy of BSC-L for all concrete logics is proved in the same way as for BSC-K$_3$. Therefore we limit our presentation to systematic characterisation of rules from which the BSC for suitable logic can be composed.

We start with rules for respective unary operations (including Łukasiewicz's modalities):

$$(\mid \neg_H\Rightarrow)\ \frac{S \mid \Pi \Rightarrow \Sigma, \varphi}{S \mid \neg\varphi, \Pi \Rightarrow \Sigma}\ (\mid\Rightarrow\neg_H)\ \frac{S \mid \varphi, \Pi \Rightarrow \Sigma}{S \mid \Pi \Rightarrow \Sigma, \neg\varphi}$$

$$(\neg_B\Rightarrow \mid)\ \frac{\Gamma \Rightarrow \Delta, \varphi \mid S}{\neg\varphi, \Gamma \Rightarrow \Delta \mid S}\ (\Rightarrow\neg_B \mid)\ \frac{\varphi, \Gamma \Rightarrow \Delta \mid S}{\Gamma \Rightarrow \Delta, \neg\varphi \mid S}$$

$$(\mid \neg_P\Rightarrow)\ \frac{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi \qquad \varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \neg\varphi, \Pi \Rightarrow \Sigma}$$

$$(\mid\Rightarrow\neg_P)\ \frac{\Gamma \Rightarrow \Delta, \varphi \mid \varphi, \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \neg\varphi}\ (\neg_{DP}\Rightarrow\mid)\ \frac{\Gamma \Rightarrow \Delta, \varphi \mid \varphi, \Pi \Rightarrow \Sigma}{\neg\varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}$$

$$(\Rightarrow \neg_{DP} \mid)\ \frac{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi \qquad \varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta, \neg\varphi \mid \Pi \Rightarrow \Sigma}$$

The remaining rules in each case (namely $(\neg_H\Rightarrow\ \mid)$, $(\Rightarrow\neg_H\ \mid)$, $(\mid\neg_B\Rightarrow)$, $(\mid\Rightarrow\neg_B)$ $(\neg_P\Rightarrow\ \mid)$, $(\Rightarrow\neg_P\ \mid)$, $(\mid\ \neg_{DP}\Rightarrow)$ and $(\mid\Rightarrow\neg_{DP})$) are like respective rules of BSC-K$_3$. Consider premisses of $(\mid\Rightarrow\neg_P)$ and $(\neg_{DP}\Rightarrow\mid)$ displaying two occurrences of the same side formula: in semantical terms it gives the effect of evaluating φ as undefined.

$$(\Diamond\!\Rightarrow |)\ \frac{\Gamma \Rightarrow \Delta \mid \varphi, \Pi \Rightarrow \Sigma}{\Diamond\varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}\quad (\Rightarrow\!\Diamond |)\ \frac{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi}{\Gamma \Rightarrow \Delta, \Diamond\varphi \mid \Pi \Rightarrow \Sigma}$$

$$(| \Diamond\!\Rightarrow)\ \frac{\Gamma \Rightarrow \Delta \mid \varphi, \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \Diamond\varphi, \Pi \Rightarrow \Sigma}\quad (|\!\Rightarrow\!\Diamond)\ \frac{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi}{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \Diamond\varphi}$$

$$(\Box\!\Rightarrow |)\ \frac{\varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}{\Box\varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}\quad (\Rightarrow\!\Box |)\ \frac{\Gamma \Rightarrow \Delta, \varphi \mid \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta, \Box\varphi \mid \Pi \Rightarrow \Sigma}$$

$$(| \Box\!\Rightarrow)\ \frac{\varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \Box\varphi, \Pi \Rightarrow \Sigma}\quad (|\!\Rightarrow\!\Box)\ \frac{\Gamma \Rightarrow \Delta, \varphi \mid \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \Box\varphi}$$

Not surprisingly rules introducing modal formula to antecedents or to succedents of 1- and 2-sequents have the same premisses; this is a consequence of the fact that such formula is never undefined. The same remark applies to rules for $\neg_H$ and $\neg_B$.

The set of rules for weak $\wedge, \vee, \rightarrow$ is also partly identical with those for BSC-K$_3$. The identical rules are $(\wedge_w\!\Rightarrow\!|)$, $(\Rightarrow\!\wedge_w |)$, $(|\!\Rightarrow\!\vee_w)$, $(| \vee_w\!\Rightarrow)$, $(|\!\Rightarrow\!\rightarrow_w)$ and $(|\!\rightarrow_w\!\Rightarrow)$. In the remaining cases we have three premiss rules:

$$(| \wedge_w\!\Rightarrow)\ \frac{\Gamma \Rightarrow \Delta \mid \varphi, \psi, \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta, \varphi \mid \varphi, \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta, \psi \mid \psi, \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \varphi \wedge \psi, \Pi \Rightarrow \Sigma}$$

$$(|\!\Rightarrow\!\wedge_w)\ \frac{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi, \psi \qquad \varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \psi \qquad \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi}{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi \wedge \psi}$$

$$(\Rightarrow\!\vee_w |)\ \frac{\Gamma \Rightarrow \Delta, \varphi, \psi \mid \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta, \varphi \mid \varphi, \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta, \psi \mid \psi, \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta, \varphi \vee \psi \mid \Pi \Rightarrow \Sigma}$$

$$(\vee_w\!\Rightarrow |)\ \frac{\varphi, \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma \qquad \varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \psi \qquad \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi}{\varphi \vee \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}$$

$$(\Rightarrow\!\rightarrow_w |)\ \frac{\Gamma \Rightarrow \Delta, \psi \mid \varphi, \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta, \varphi \mid \varphi, \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta, \psi \mid \psi, \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta, \varphi \rightarrow \psi \mid \Pi \Rightarrow \Sigma}$$

$$(\rightarrow_w\!\Rightarrow |)\ \frac{\varphi, \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi, \psi \qquad \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi}{\varphi \rightarrow \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}$$

In the case of $\mathbf{K}_3^{\rightarrow}$ and $\mathbf{K}_3^{\leftarrow}$ the specific rules are:

$$(| \Rightarrow\!\wedge_{mC})\ \frac{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi \qquad \varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \psi}{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi \wedge \psi}$$

$$(| \wedge\!\Rightarrow_{mC})\ \frac{\Gamma \Rightarrow \Delta \mid \varphi, \psi, \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta, \varphi \mid \varphi, \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \varphi \wedge \psi, \Pi \Rightarrow \Sigma}$$

$$(\vee\!\Rightarrow_{mC} |)\ \frac{\varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma \qquad \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi}{\varphi \vee \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}$$

$$(\Rightarrow\!\vee_{mC} |)\ \frac{\Gamma \Rightarrow \Delta, \varphi, \psi \mid \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta, \varphi \mid \varphi, \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta, \varphi \vee \psi \mid \Pi \Rightarrow \Sigma}$$

$$(\Rightarrow\!\rightarrow_{mC} |)\ \frac{\Gamma \Rightarrow \Delta, \psi \mid \varphi, \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta, \varphi \mid \varphi, \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta, \varphi \rightarrow \psi \mid \Pi \Rightarrow \Sigma}$$

$$(\to\Rightarrow_{mC}|)\quad \frac{\varphi,\psi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma \qquad \varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma,\varphi}{\varphi \to \psi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma}$$

$$(|\Rightarrow\wedge_K)\quad \frac{\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma,\psi \qquad \psi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma,\varphi}{\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma,\varphi \wedge \psi}$$

$$(|\wedge\Rightarrow_K)\quad \frac{\varGamma \Rightarrow \varDelta \mid \varphi,\psi,\varPi \Rightarrow \varSigma \qquad \varGamma \Rightarrow \varDelta,\psi \mid \psi,\varPi \Rightarrow \varSigma}{\varGamma \Rightarrow \varDelta \mid \varphi \wedge \psi,\varPi \Rightarrow \varSigma}$$

$$(\vee\Rightarrow_K|)\quad \frac{\varphi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma,\psi \qquad \psi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma}{\varphi \vee \psi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma}$$

$$(\Rightarrow\vee_K|)\quad \frac{\varGamma \Rightarrow \varDelta,\varphi,\psi \mid \varPi \Rightarrow \varSigma \qquad \varGamma \Rightarrow \varDelta,\psi \mid \psi,\varPi \Rightarrow \varSigma}{\varGamma \Rightarrow \varDelta,\varphi \vee \psi \mid \varPi \Rightarrow \varSigma}$$

$$(\Rightarrow\to_K|)\quad \frac{\varGamma \Rightarrow \varDelta,\psi \mid \varphi,\varPi \Rightarrow \varSigma \qquad \varGamma \Rightarrow \varDelta,\psi \mid \psi,\varPi \Rightarrow \varSigma}{\varGamma \Rightarrow \varDelta,\varphi \to \psi \mid \varPi \Rightarrow \varSigma}$$

$$(\to\Rightarrow_K|)\quad \frac{\psi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma \qquad \varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma,\varphi,\psi}{\varphi \to \psi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma}$$

The remaining rules in both cases are identical with $(\wedge\Rightarrow|)$, $(\Rightarrow\wedge\ |)$, $(|\Rightarrow\vee)$, $(|\ \vee\Rightarrow)$, $(|\ \Rightarrow\to)$ and $(|\ \to\Rightarrow)$ from BSC-K$_3$.

The implication of Łukasiewicz [34] and his specific additive $\wedge_L$ and $\vee_L$ are characterised by the following rules:

$$(|\ \wedge_L \Rightarrow)\quad \frac{\varphi,\varGamma \Rightarrow \varDelta \mid \psi,\varPi \Rightarrow \varSigma \qquad \psi,\varGamma \Rightarrow \varDelta \mid \varphi,\varPi \Rightarrow \varSigma}{\varGamma \Rightarrow \varDelta \mid \varphi \wedge \psi,\varPi \Rightarrow \varSigma}$$

$$(|\Rightarrow \wedge_L)\quad \frac{\varGamma \Rightarrow \varDelta,\varphi,\psi \mid \varPi \Rightarrow \varSigma \qquad \varphi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma,\psi \qquad \psi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma,\varphi}{\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma,\varphi \wedge \psi}$$

$$(|\ \vee_L \Rightarrow)\quad \frac{\varphi,\varGamma \Rightarrow \varDelta \mid \psi,\varPi \Rightarrow \varSigma \qquad \psi,\varGamma \Rightarrow \varDelta \mid \varphi,\varPi \Rightarrow \varSigma}{\varGamma \Rightarrow \varDelta \mid \varphi \vee \psi,\varPi \Rightarrow \varSigma}$$

$$(|\Rightarrow \vee_L)\quad \frac{\varGamma \Rightarrow \varDelta,\varphi,\psi \mid \varPi \Rightarrow \varSigma \qquad \varphi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma,\psi \qquad \psi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma,\varphi}{\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma,\varphi \vee \psi}$$

$$(\Rightarrow\to_L|)\quad \frac{\varphi,\varGamma \Rightarrow \varDelta,\psi \mid \varPi \Rightarrow \varSigma \qquad \varGamma \Rightarrow \varDelta \mid \varphi,\varPi \Rightarrow \varSigma,\psi}{\varGamma \Rightarrow \varDelta,\varphi \to \psi \mid \varPi \Rightarrow \varSigma}$$

$$(\to_L\Rightarrow|)\quad \frac{\varGamma \Rightarrow \varDelta,\varphi \mid \psi,\varPi \Rightarrow \varSigma \qquad \psi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma \qquad \varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma,\varphi}{\varphi \to \psi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma}$$

The remaining rules are identical with $(\wedge\Rightarrow|)$, $(\Rightarrow\wedge\ |)$, $(\Rightarrow\vee\ |)$, $(\vee\Rightarrow|)$, $(|\ \Rightarrow\to)$ and $(|\ \to\Rightarrow)$ from BSC-K$_3$.

For Sobociński's connectives we have:

$$(\wedge_S \Rightarrow|)\quad \frac{\varphi,\varGamma \Rightarrow \varDelta \mid \psi,\varPi \Rightarrow \varSigma \qquad \psi,\varGamma \Rightarrow \varDelta \mid \varphi,\varPi \Rightarrow \varSigma}{\varphi \wedge \psi,\varGamma \Rightarrow \varDelta \mid \varPi \Rightarrow \varSigma}$$

$$(\Rightarrow \wedge_S \mid)\ \frac{\Gamma \Rightarrow \Delta, \varphi, \psi \mid \Pi \Rightarrow \Sigma \qquad \varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \psi \qquad \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi}{\Gamma \Rightarrow \Delta, \varphi \wedge \psi \mid \Pi \Rightarrow \Sigma}$$

$$(\mid\Rightarrow\vee_S)\ \frac{\Gamma \Rightarrow \Delta, \varphi \mid \Pi \Rightarrow \Sigma, \psi \qquad \Gamma \Rightarrow \Delta, \psi \mid \Pi \Rightarrow \Sigma, \varphi}{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi \vee \psi}$$

$$(\mid \vee_S\Rightarrow)\ \frac{\Gamma \Rightarrow \Delta \mid \varphi, \psi, \Pi \Rightarrow \Sigma \qquad \varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \psi \qquad \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi}{\Gamma \Rightarrow \Delta \mid \varphi \vee \psi, \Pi \Rightarrow \Sigma}$$

$$(\mid\Rightarrow\rightarrow_S)\ \frac{\varphi, \Gamma \Rightarrow \Delta, \psi \mid \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta \mid \varphi, \Pi \Rightarrow \Sigma, \psi}{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi \rightarrow \psi}$$

$$(\mid\rightarrow_S\Rightarrow)\ \frac{\Gamma \Rightarrow \Delta, \varphi \mid \psi, \Pi \Rightarrow \Sigma \qquad \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi}{\Gamma \Rightarrow \Delta \mid \varphi \rightarrow \psi, \Pi \Rightarrow \Sigma}$$

The remaining rules look like in BSC-K$_3$.

Sette's connectives are characterised by the following rules:

$$(\Rightarrow\wedge_{Se} \mid)\ \frac{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi \qquad \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \psi}{\Gamma \Rightarrow \Delta, \varphi \wedge \psi \mid \Pi \Rightarrow \Sigma}$$

$$(\wedge_{Se}\Rightarrow\mid)\ \frac{\Gamma \Rightarrow \Delta \mid \varphi, \psi, \Pi \Rightarrow \Sigma}{\varphi \wedge \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}\ (\Rightarrow\vee_{Se} \mid)\ \frac{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi, \psi}{\Gamma \Rightarrow \Delta, \varphi \vee \psi \mid \Pi \Rightarrow \Sigma}$$

$$(\vee_{Se}\Rightarrow\mid)\ \frac{\Gamma \Rightarrow \Delta \mid \varphi, \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta \mid \psi, \Pi \Rightarrow \Sigma}{\varphi \vee \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}$$

$$(\rightarrow_{Se}\Rightarrow\mid)\ \frac{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi \qquad \Gamma \Rightarrow \Delta \mid \psi, \Pi \Rightarrow \Sigma}{\varphi \rightarrow \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}$$

$$(\Rightarrow\rightarrow_{Se}\mid)\ \frac{\Gamma \Rightarrow \Delta \mid \varphi, \Pi \Rightarrow \Sigma, \psi}{\Gamma \Rightarrow \Delta, \varphi \rightarrow \psi \mid \Pi \Rightarrow \Sigma}\ (\mid\Rightarrow\rightarrow_{Se})\ \frac{S \mid \varphi, \Pi \Rightarrow \Sigma, \psi}{S \mid \Pi \Rightarrow \Sigma, \varphi \rightarrow \psi}$$

$$(\mid\rightarrow_{Se}\Rightarrow)\ \frac{S \mid \Pi \Rightarrow \Sigma, \varphi \qquad S \mid \psi, \Pi \Rightarrow \Sigma}{S \mid \varphi \rightarrow \psi, \Pi \Rightarrow \Sigma}$$

$(\mid \wedge_{Se}\Rightarrow)$, $(\mid\Rightarrow\wedge_{Se})$, $(\mid\Rightarrow\vee_{Se})$, $(\mid \vee_{Se}\Rightarrow)$ are like in BSC-K$_3$.

Finally Carnielli and Sette connectives characterising $\mathbf{I^1}$ and $\mathbf{I^2}$:

$$(\mid\Rightarrow\wedge_C)\ \frac{\Gamma \Rightarrow \Delta, \varphi \mid \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta, \psi \mid \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi \wedge \psi}$$

$$(\mid\wedge_C\Rightarrow)\frac{\varphi, \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \varphi \wedge \psi, \Pi \Rightarrow \Sigma}\ (\mid\Rightarrow\vee_C)\frac{\Gamma \Rightarrow \Delta, \varphi, \psi \mid \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi \vee \psi}$$

$$(|\vee_C\Rightarrow)\ \frac{\varphi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma \quad \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \varphi \vee \psi, \Pi \Rightarrow \Sigma}$$

$$(|\to_C\Rightarrow)\ \frac{\Gamma \Rightarrow \Delta, \varphi \mid \Pi \Rightarrow \Sigma \quad \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \varphi \to \psi, \Pi \Rightarrow \Sigma}$$

$$(|\Rightarrow\to_C)\frac{\varphi, \Gamma \Rightarrow \Delta, \psi \mid \Pi \Rightarrow \Sigma}{\Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi \to \psi}\ (\Rightarrow\to_C|)\ \frac{\varphi, \Gamma \Rightarrow \Delta, \psi \mid S}{\Gamma \Rightarrow \Delta, \varphi \to \psi \mid S}$$

$$(\to_C\Rightarrow|)\ \ \frac{\Gamma \Rightarrow \Delta, \varphi \mid S \qquad \psi, \Gamma \Rightarrow \Delta \mid S}{\varphi \to \psi, \Gamma \Rightarrow \Delta \mid S}$$

$(\wedge_C\Rightarrow|)$, $(\Rightarrow\wedge_C|)$, $(\Rightarrow\vee_C|)$, $(\vee_C\Rightarrow|)$ are like in BSC-K$_3$.

We finish with the characterisation of the remaining implications introduced in Sect. 2. In most cases it is obtained by combining rules which were previously introduced. In particular:

Słupecki's [49] implication is characterised by means of: $(|\Rightarrow\to)$ and $(|\to\Rightarrow)$ from BSC-K$_3$ as well as $(\Rightarrow\to_C|)$ and $(\to_C\Rightarrow|)$.

Heyting's implication [22] is characterised by means of: $(\to_L\Rightarrow|)$, $(\Rightarrow\to_L|)$, $(|\Rightarrow\to_{Se})$,$(|\to_{Se}\Rightarrow)$.

D'Ottaviano/DaCosta/Jaśkowski/Słupecki's implication [13,30,48] is characterised by means of: $(\to\Rightarrow|)$, $(\Rightarrow\to|)$, $(|\Rightarrow\to_{Se})$,$(|\to_{Se}\Rightarrow)$.

Rescher's implication [43] is characterised by means of: $(\to_L\Rightarrow|)$, $(\Rightarrow\to_L|)$, $(|\Rightarrow\to_S)$,$(|\to_S\Rightarrow)$.

Tomova's implication [22] is characterised by means of: $(\to_L\Rightarrow|)$, $(\Rightarrow\to_L|)$, $(|\Rightarrow\to_C)$,$(|\to_C\Rightarrow)$.

Only in case of Sobociński's implication $\to'_S$ we have a pair of new rules:

$$(\to'_S\Rightarrow|)\ \ \frac{\psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma \qquad \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma, \varphi, \psi \qquad \Gamma \Rightarrow \Delta, \varphi \mid \varphi, \psi, \Pi \Rightarrow \Sigma}{\varphi \to \psi, \Gamma \Rightarrow \Delta \mid \Pi \Rightarrow \Sigma}$$

$$(\Rightarrow\to'_S|)\ \ \frac{\varphi, \Gamma \Rightarrow \Delta, \psi \mid \Pi \Rightarrow \Sigma \quad \Gamma \Rightarrow \Delta \mid \varphi, \Pi \Rightarrow \Sigma, \psi \quad \Gamma \Rightarrow \Delta, \psi \mid \psi, \Pi \Rightarrow \Sigma, \varphi}{\Gamma \Rightarrow \Delta, \varphi \to \psi \mid \Pi \Rightarrow \Sigma}$$

The remaining two rules are: $(|\Rightarrow\to_{Se})$ and $(|\to_{Se}\Rightarrow)$.

Let us show how $(\Rightarrow\to'_S|)$ was obtained on the basis of the table for $\to'_S$ from p. 4. $\varphi \to \psi$ is either false or undefined which corresponds to four cells:

$\to'_S$	1	u	0	
1		u	0	this row says that φ is 1 and ψ is 0 or u – the left premiss;
u			0	
0		u		this row says that φ is 0 and ψ is u – the right premiss.

The remaining premiss covers the cell with 0 in the first and second rows attributed to ψ while φ is 1 or u. Note that since the left premiss covers the first row and the right premiss covers the last row we could alternatively formulate the middle premiss as $\Gamma \Rightarrow \Delta, \varphi \mid \varphi, \Pi \Rightarrow \Sigma, \psi$ to cover exactly the cell with 0 in the second row (here φ is just u) but since ψ is 0 in two rows where φ is 1 or u we can be more economical here. A reader can check that many rules can be formulated in alternative way. We always tried to find the most economical representation which can be used easily also for proving syntactically the cut elimination theorem (which will be shown in the extended version of this paper).

Now, consider an arbitrary connective c of the logic **L**, the corresponding operation $\underline{c}$ as characterised by suitable matrix determining **L** in Sect. 2, and the four rules for c. It holds:

Theorem 4. *For all presented rules characterising arbitrary c of any* **L***: all premisses are* **L***-valid iff the conclusion is* **L***-valid.*

Proof. This is an analogue of Theorem 1 for any considered logic **L** which implies adequacy of respective BSC-L. □

5 Interpolation

We present a constructive proof of the interpolation theorem for some logics based on the strategy proposed by Muskens and Wintein [58]. It was originally applied in tableau setting for Belnap-Dunn four-valued logic as well as for $\mathbf{K_3}$ and **LP**. Here we demonstrate that BSC can be also used for showing that interpolation holds for some paracomplete and paraconsistent logics. Let $\mathbf{L} \in \{\mathbf{I^1}, \mathbf{I^2}, \mathbf{P^1}, \mathbf{P^2}\}$.

Theorem 5. *For any contingent formulae φ, ψ, if $\varphi \models_L \psi$, then we can construct an interpolant for $\mathbf{I^1}, \mathbf{I^2}$ on the basis of proof-search trees for $\varphi \Rightarrow \mid \Rightarrow$ and $\Rightarrow \psi \mid \Rightarrow$ and an interpolant for $\mathbf{P^1}, \mathbf{P^2}$ on the basis of proof-search trees for $\Rightarrow \mid \varphi \Rightarrow$ and $\Rightarrow \mid \Rightarrow \psi$ in suitable BSC-L.*

Proof. We will demonstrate the case of BSC-I^1; the case of BSC-I^2 is identical and the cases of BSC-P^1 and BSC-P^2 are dual, so we only comment on them in the key points. Assume that $\varphi \models_{I^1} \psi$; hence by completeness we have a cut-free proof of $\varphi \Rightarrow \psi \mid \Rightarrow$ in BSC-I^1. Now produce complete proof-search trees for $\varphi \Rightarrow \mid \Rightarrow$ and $\Rightarrow \psi \mid \Rightarrow$. Since φ, ψ are contingent, they have some non-axiomatic leaves. Let $\Gamma_1 \Rightarrow \Delta_1 \mid \Pi_1 \Rightarrow \Sigma_1, \ldots, \Gamma_k \Rightarrow \Delta_k \mid \Pi_k \Rightarrow \Sigma_k$ be the list of non-axiomatic atomic leaves of the proof-search tree for $\varphi \Rightarrow \mid \Rightarrow$ and $\Theta_1 \Rightarrow \Lambda_1 \mid \Xi_1 \Rightarrow \Omega_1, \ldots, \Theta_n \Rightarrow \Lambda_n \mid \Xi_n \Rightarrow \Omega_n$ such a list taken from the proof-search tree for $\Rightarrow \psi \mid \Rightarrow$. It holds:

Claim (1). For any $i \leq k$ and $j \leq n$, $\Gamma_i, \Theta_j \Rightarrow \Delta_i, \Lambda_j \mid \Pi_i, \Xi_j \Rightarrow \Sigma_i, \Omega_j$ is an axiomatic atomic bisequent.

To see this take a tree for $\varphi \Rightarrow \mid \Rightarrow$ and add ψ to succedents of all 1-sequents in the tree. Due to context independence of all rules it is a correct proof-search tree. Now for each leaf $\Gamma_i \Rightarrow \Delta_i, \psi \mid \Pi_i \Rightarrow \Sigma_i$ append a tree of $\Rightarrow \psi \mid \Rightarrow$ but with Γ_i added to each antecedent and Δ_i added to each succedent of 1-sequents, and similarly with Π_i and Σ_i in all 2-sequents. In the resulting proof-search tree we have leaves of the form $\Gamma_i, \Theta_j \Rightarrow \Delta_i, \Lambda_j \mid \Pi_i, \Xi_j \Rightarrow \Sigma_i, \Omega_j$ for all $i \leq k$ and $j \leq n$. If at least one of them is not axiomatic, then $\nvdash \varphi \Rightarrow \psi \mid \Rightarrow$. □

Next for every $\Gamma_i \Rightarrow \Delta_i \mid \Pi_i \Rightarrow \Sigma_i, i \leq k$, define the following sets:

$\Gamma_i' = \Gamma_i \cap (\bigcup \Lambda_j \cup \bigcup \Omega_j)$ for $j \leq n$
$\Delta_i' = \Delta_i \cap \bigcup \Theta_j$ for $j \leq n$
$\Pi_i' = \Pi_i \cap \bigcup \Omega_j$ for $j \leq n$
$\Sigma_i' = \Sigma_i \cap (\bigcup \Theta_j \cup \bigcup \Xi_j)$ for $j \leq n$

Since every $\Gamma_i, \Theta_j \Rightarrow \Delta_i, \Lambda_j \mid \Pi_i, \Xi_j \Rightarrow \Sigma_i, \Omega_j$ is axiomatic we are guaranteed that $\Gamma_i' \cup \Delta_i' \cup \Pi_i' \cup \Sigma_i' \neq \varnothing$. Note also that $AT(\Gamma_i' \cup \Delta_i' \cup \Pi_i' \cup \Sigma_i') \subseteq AT(\varphi) \cap AT(\psi)$, where AT stands for the set of atoms. Now define an interpolant $Int(\varphi, \psi)$ for considered logics. For $\mathbf{I^1}, \mathbf{I^2}$ it has the same form:

$$\bigwedge \Gamma_1' \wedge \bigwedge \neg\Sigma_1' \wedge \neg\big(\bigvee \neg\Pi_1' \vee \bigvee \Delta_1'\big) \vee \ldots \vee \bigwedge \Gamma_k' \wedge \bigwedge \neg\Sigma_k' \wedge \neg\big(\bigvee \neg\Pi_k' \vee \bigvee \Delta_k'\big),$$

where $\neg\Pi$ means the set of negations of all elements in Π.

For $\mathbf{P^1}, \mathbf{P^2}$ $Int(\varphi, \psi)$ is defined as:

$$\bigwedge \Pi_1' \wedge \bigwedge \neg\Delta_1' \wedge \neg\big(\bigvee \neg\Gamma_1' \vee \bigvee \Sigma_1'\big) \vee \ldots \vee \bigwedge \Pi_k' \wedge \bigwedge \neg\Delta_k' \wedge \neg\big(\bigvee \neg\Gamma_k' \vee \bigvee \Sigma_k'\big)$$

We can show that:

Claim (2). $Int(\varphi, \psi)$ is an interpolant for $\varphi \models_L \psi$.

Proof. As an example, we present the proof for BSC-I^1. For the sake of proof let us recall that BSC-I^1 consists of the rules characterising $\wedge_C, \vee_C, \rightarrow_C$ and $\neg_H$. However, most of the rules necessary for conducting the proof are identical with respective rules from BSC-K$_3$, so the label C in their names will be omitted in these cases for easier recognition where the specific rules (concretely $(\mid \vee_C \Rightarrow)$ and $(\mid\Rightarrow \vee_C)$) are required.

Since for every $\bigwedge \Gamma_i' \wedge \bigwedge \neg\Sigma_i' \wedge \neg(\bigvee \neg\Pi_i' \vee \bigvee \Delta_i')$ all (negated) atoms are by definition taken from $AT(\varphi) \cap AT(\psi)$, we must only prove that BSC-I^1 $\vdash \varphi \Rightarrow Int(\varphi, \psi) \mid \Rightarrow$, and BSC-I^1 $\vdash Int(\varphi, \psi) \Rightarrow \psi \mid \Rightarrow$ (the same for BSC-I^2), and BSC-P^1 $\vdash \Rightarrow \mid \varphi \Rightarrow Int(\varphi, \psi)$ and BSC-P^1 $\vdash \Rightarrow \mid Int(\varphi, \psi) \Rightarrow \psi$ (and the same for BSC-P^2).

Again take a complete proof-search tree for $\varphi \Rightarrow \mid \Rightarrow$ and add $Int(\varphi, \psi)$ to every succedent of 1-sequent. For every $\Gamma_i \Rightarrow \Delta_i, Int(\varphi, \psi) \mid \Pi_i \Rightarrow \Sigma_i$ apply $(\Rightarrow \vee \mid)$ to get

$$\Gamma_i \Rightarrow \Delta_i, \bigwedge \Gamma_i' \wedge \bigwedge \neg\Sigma_i' \wedge \neg(\bigvee \neg\Pi_i' \vee \bigvee \Delta_i'), Int(\varphi, \psi)^{-i} \mid \Pi_i \Rightarrow \Sigma_i,$$

where $Int(\varphi, \psi)^{-i}$ is the rest of the disjunction (if any). Applying $(\Rightarrow \wedge \mid)$ we obtain three bisequents:

(a) $\Gamma_i \Rightarrow \Delta_i, \bigwedge \Gamma'_i, \mathit{Int}(\varphi, \psi)^{-i} \mid \Pi_i \Rightarrow \Sigma_i$
(b) $\Gamma_i \Rightarrow \Delta_i, \bigwedge \neg\Sigma'_i, \mathit{Int}(\varphi, \psi)^{-i} \mid \Pi_i \Rightarrow \Sigma_i$
(c) $\Gamma_i \Rightarrow \Delta_i, \neg(\bigvee \neg\Pi'_i \vee \bigvee \Delta'_i), \mathit{Int}(\varphi, \psi)^{-i} \mid \Pi_i \Rightarrow \Sigma_i$.

Systematically applying ($\Rightarrow \wedge \mid$) to (a) we obtain $\Gamma_i \Rightarrow \Delta_i, p, \mathit{Int}(\varphi, \psi)^{-i} \mid \Pi_i \Rightarrow \Sigma_i$ for each $p \in \Gamma'_i$ and since $\Gamma'_i \subseteq \Gamma_i$ they are all axiomatic. Similarly with (b) but now we first obtain $\Gamma_i \Rightarrow \Delta_i, \neg p, \mathit{Int}(\varphi, \psi)^{-i} \mid \Pi_i \Rightarrow \Sigma_i$ for each $p \in \Sigma'_i$. After the application of ($\Rightarrow \neg \mid$) we obtain $\Gamma_i \Rightarrow \Delta_i, \mathit{Int}(\varphi, \psi)^{-i} \mid p, \Pi_i \Rightarrow \Sigma_i$ which is axiomatic since $\Sigma'_i \subseteq \Sigma_i$. For (c) we first apply ($\Rightarrow \neg \mid$) and obtain $\Gamma_i \Rightarrow \Delta_i, \mathit{Int}(\varphi, \psi)^{-i} \mid \bigvee \neg\Pi'_i \vee \bigvee \Delta'_i, \Pi_i \Rightarrow \Sigma_i$. By ($\mid \vee_C \Rightarrow$) we obtain: $\bigvee \neg\Pi'_i, \Gamma_i \Rightarrow \Delta_i, \mathit{Int}(\varphi, \psi)^{-i} \mid \Pi_i \Rightarrow \Sigma_i$ and $\bigvee \Delta'_i, \Gamma_i \Rightarrow \Delta_i, \mathit{Int}(\varphi, \psi)^{-i} \mid \Pi_i \Rightarrow \Sigma_i$. Systematic application of ($\vee \Rightarrow \mid$) to the latter produces axiomatic bisequents $p, \Gamma_i \Rightarrow \Delta_i, \mathit{Int}(\varphi, \psi)^{-i} \mid \Pi_i \Rightarrow \Sigma_i$ for each $p \in \Delta'_i$. Systematic application of ($\vee \Rightarrow \mid$) to the former produces $\neg p, \Gamma_i \Rightarrow \Delta_i, \mathit{Int}(\varphi, \psi)^{-i} \mid \Pi_i \Rightarrow \Sigma_i$ for each $p \in \Pi'_i$. After application of ($\neg \Rightarrow \mid$) they also yield axiomatic sequents. Hence we have a proof of $\varphi \Rightarrow \mathit{Int}(\varphi, \psi) \mid \Rightarrow$.

We have to do the same with a complete proof-search tree for $\Rightarrow \psi \mid \Rightarrow$ but now adding $\mathit{Int}(\varphi, \psi)$ to every antecedent of all 1-sequents in the tree. For every leaf $\mathit{Int}(\varphi, \psi), \Theta_j \Rightarrow \Lambda_j \mid \Xi_j \Rightarrow \Omega_j$ we apply ($\vee \Rightarrow \mid$) to each disjunct of $\mathit{Int}(\varphi, \psi)$ until we get leaves: $\bigwedge \Gamma'_1 \wedge \bigwedge \neg\Sigma'_1 \wedge \neg(\bigvee \neg\Pi'_1 \vee \bigvee \Delta'_1), \Theta_j \Rightarrow \Lambda_j \mid \Xi_j \Rightarrow \Omega_j \ldots \bigwedge \Gamma'_k \wedge \bigwedge \neg\Sigma'_k \wedge \neg(\bigvee \neg\Pi'_k \vee \bigvee \Delta'_k), \Theta_j \Rightarrow \Lambda_j \mid \Xi_j \Rightarrow \Omega_j$. To each such leaf we apply ($\wedge \Rightarrow \mid$) obtaining bisequents of the form $\Gamma'_i, \neg\Sigma'_i, \neg(\bigvee \neg\Pi'_i \vee \bigvee \Delta'_i), \Theta_j \Rightarrow \Lambda_j \mid \Xi_j \Rightarrow \Omega_j$ for $i \leq k, j \leq n$. In each case the application of ($\neg \Rightarrow \mid$) yields $\Gamma'_i, \Theta_j \Rightarrow \Lambda_j \mid \Xi_j \Rightarrow \Omega_j, \Sigma'_i, \bigvee \neg\Pi'_i \vee \bigvee \Delta'_i$. The application of ($\mid \Rightarrow \vee_C$) to $\bigvee \neg\Pi'_i \vee \bigvee \Delta'_i$ yields $\Gamma'_i, \Theta_j \Rightarrow \Lambda_j, \bigvee \neg\Pi'_i, \bigvee \Delta'_i \mid \Xi_j \Rightarrow \Omega_j, \Sigma'_i$. Systematic application of ($\Rightarrow \vee \mid$) and ($\Rightarrow \neg \mid$) gives leaves of the form $\Gamma'_i, \Theta_j \Rightarrow \Lambda_j, \Delta'_i \mid \Pi'_i, \Xi_j \Rightarrow \Omega_j, \Sigma'_i$. Since for every $i \leq k, j \leq n$, $\Gamma_i, \Theta_j \Rightarrow \Delta_i, \Lambda_j \mid \Pi_i, \Xi_j \Rightarrow \Sigma_i, \Omega_j$ is axiomatic these primed versions are axiomatic too. Assume the contrary, then it must be e.g. some $p \notin \Gamma'_i$ such that either $p \in \Gamma_i \cap \Lambda_j$ or $p \in \Gamma_i \cap \Omega_j$ (or for other pairs generating axioms). But it is impossible since by definition Γ'_i must contain such p (and the same for other cases of primed sets). □

The proof for BSC-I^2 is identical since the only difference between these two logics is that $\mathbf{I^1}$ has Heyting's negation whereas in $\mathbf{I^2}$ it is Kleene's negation. But the two BSC rules for negation which are used in the proof are common to both negations. The proof for $\mathbf{P^1}, \mathbf{P^2}$ is dual to the above and uses slightly different definition of $\mathit{Int}(\varphi, \psi)$ specified above. Again the two logics differ only with respect to negations, but the rules used in the proof are common to Bochvar's and Kleene's one.

Eventually note that this proof may be applied also to other logics but in some cases it is convenient to extend their languages. For example, interpolants for some logics can be defined as disjunctions of the following formulae:

For $\mathbf{K_3}$ – $\bigwedge \Gamma'_i \wedge \bigwedge \neg\Sigma'_i \wedge \bigwedge \neg_B \Delta'_i \wedge \bigwedge \neg_B \neg\Pi'_i$
For **LP** – $\bigwedge \Pi'_i \wedge \bigwedge \neg\Delta'_i \wedge \bigwedge \neg_H \Sigma'_i \wedge \bigwedge \neg_H \neg\Gamma'_i$
For $\mathbf{G_3}$ – $\bigwedge \Gamma'_i \wedge \neg_H(\bigwedge \Pi'_i \rightarrow \bigvee \Sigma'_i) \wedge \bigwedge \neg_B \Delta'_i$
For $\mathbf{G'_3}$ – $\bigwedge \Pi'_i \wedge \bigwedge \neg_B \Delta'_i \wedge \bigwedge \neg_H \Sigma'_i \wedge \bigwedge \neg_B \neg_B \Gamma'_i$

6 Conclusion

Bisequent calculi can be seen as one of the possible syntactical realizations of so called Suszko's thesis [54] in the treatment of many-valued logics. According to Suszko every logic is two-valued in the sense that all values are divided into designated and non-designated and this is reflected in the definition of consequence relation. In the case of bisequent calculi it is additionally made evident that two possible choices of designated values can be made. However, on a deep level a BSC is similar to some other proposed formalisations mentioned in the Introduction. On one hand bisequents resemble several labelled approaches where labels denote sets of values; a difference is that instead of labels a position of a formula in a bisequent is crucial, hence the method is strictly syntactical. On the other hand, there is a similarity with Avron's [4] and Avron, Ben-Naim, and Konikowska's [6] sequent calculi with special rules defined for negated formulae; a difference is that BSC satisfies ordinary subformula property and purity conditions to the effect that in schemata of rules only one (occurrence of a) connective is involved. The price is that instead of standard sequents we use a pair of them.

As we mentioned in the Introduction there is one more general difference. In the case of labelled calculi or Avron's SC we have the same input for 1- and 2-logics, whereas in BSC a different input for both classes of logics is defined; a 1- or a 2-sequent in a bisequent. A consequence of our choice is that for every pair of 1- and 2-logic with the same connectives (like e.g. $\mathbf{K}_3$ and **LP**) the rules and axioms are identical. In contrast, in other mentioned approaches for such pairs of related logics, the respective calculi must differ either with respect to some axioms (closure conditions in tableaux) or to rules. It seems that the present solution where systems differ only with respect to the input is more economical and uniform. In fact we can consider also logics determined by different notions of consequence relations while still keeping the rules and axioms intact. Two relations considered in the text express informally the situation where either truth is preserved or non-falsity is preserved. But two other possibilities are open as well: $\Gamma \Rightarrow \mid \Rightarrow \psi$ corresponds to the notion of no-counterexample consequence (see e.g. Lehmann [33], Paoli [39]), whereas $\Rightarrow \varphi \mid \Gamma \Rightarrow$ corresponds to the liberal consequence which leads from non-falsity to truth. This level of uniformity follows from the fact that rules of BSC are not computed on the basis of any normal (disjunctive or conjunctive) form, like in other approaches, but on the basis of geometrical insights illustrated in Sect. 4.

Finally notice that the application of BSC may be extended easily to first-order languages. It is quite obvious how to define suitable rules for quantifiers. But the proof of adequacy requires more refined methods than those applied here so for the lack of space we limited ourselves to propositional case. However, we finish the paper with one more problem for further investigation: the application of first-order BSC to formalisation of neutral free logics, and in particular to specific theories of definite descriptions based on some Fregean ideas (see e.g. Lehmann [33], Stenlund [52]). Since sequent and tableau calculi for such theories built on positive and negative free logics were already provided in [24,26,28], this paper offers a proper ground for extension of these results to neutral free logics.

References

1. Anderson, A.R., Belnap, N.D.: Entailment. The Logic of Relevance and Necessity, vol. 1. Princeton University Press (1975)
2. Asenjo, F.G.: A calculus of antinomie. Notre Dame J. Formal Logic **7**(1), 103–105 (1966)
3. Asenjo, F.G., Tamburino, J.: Logic of antinomies. Notre Dame J. Formal Logic **16**(1), 17–44 (1975)
4. Avron, A.: Natural 3-valued logics – characterization and proof theory. J. Symb. Logic **61**(1), 276–294 (1991)
5. Avron, A.: On an implicational connective of RM. Notre Dame J. Formal Logic **27**(2), 201–209 (1986)
6. Avron, A., Ben-Naim, J., Konikowska, B.: Cut-free ordinary sequent calculi for logics having generalized finite-valued semantics. Log. Univers. **1**(1), 41–70 (2007)
7. Batens, D.: Paraconsistent extensional propositional logics. Logique et Anal. (N.S.) **23**(90–91), 195–234 (1980)
8. Baaz, M., Fermüller, C.G., Zach, R.: Elimination of cuts in first-order finite-valued logics. J. Inf. Process. Cybern. **29**(6), 333–355 (1994)
9. Bochvar, D.A.: On a three-valued logical calculus and its application to the analysis of the paradoxes of the classical extended functional calculus (English translation of Bochvar's paper of 1938). Hist. Philos. Logic **2**(1–2), 87–112 (1981)
10. Caleiro, C., Marcelino, S.: Analytic calculi for monadic PNmatrices. In: Iemhoff, R., Moortgat, M., de Queiroz, R. (eds.) WoLLIC 2019. LNCS, vol. 11541, pp. 84–98. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-662-59533-6_6
11. Carnielli, W.A.: On sequents and tableaux for many-valued logics. J. Non-Classical Logic **8**(1), 59–76 (1991)
12. Carnielli, W.A., Marcos, J.: Taxonomy of C-systems. In: Carnielli, W.A., Coniglio, M.E., D'Ottaviano, I.M.L. (eds.) Paraconsistency: The Logical Way to the Inconsistent, pp. 1–94. CRC Press (2002)
13. D'Ottaviano, I.M.L., da Costa, N.C.A.: Sur un probléme de Jaśkowski. Comptes Rendus Acad. Sci. Paris **270A**, 1349–1353 (1970)
14. da Costa, N.C.A.: On the theory of inconsistent formal systems. Notre Dame J. Formal Logic **15**(4), 497–510 (1974)
15. Doherty, P.: A constraint-based approach to proof-procedures for multi-valued logics. In: Proceedings of the 1st World Conference on Fundamentals of Artificial Intelligence (WOCFAI). Springer (1991)
16. Fitting, M.: Kleene's three valued logics and their children. Fund. Inform. **20**(1), 113–131 (1994)
17. Gödel, K.: Zum intuitionistischen Aussgenkalkül. Anz. Akad. Wiss. Wien. **69**, 65–66 (1932)
18. Grätz, L.: Analytic tableaux for non-deterministic semantics. In: Das, A., Negri, S. (eds.) TABLEAUX 2021. LNCS (LNAI), vol. 12842, pp. 38–55. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86059-2_3
19. Hähnle, R.: Automated Deduction in Multiple-Valued Logics. Oxford University Press, Oxford (1994)
20. Hähnle, R.: Tableaux and related methods. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 101–177. Elsevier (2001)
21. Halldén, S.: The logic of nonsense. Lundequista Bokhandeln (1949)
22. Heyting, A.: Die Formalen Regeln der intuitionistischen Logik. Sitzungsber. Preussischen Acad. Wiss. Berlin 42–46 (1930)

23. Indrzejczak, A.: Two is enough – bisequent calculus for S5. In: Herzig, A., Popescu, A. (eds.) FroCoS 2019. LNCS (LNAI), vol. 11715, pp. 277–294. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29007-8_16
24. Indrzejczak, A.: Free definite description theory - sequent calculi and cut elimination. Logic Log. Philos. **29**(4), 505–539 (2020)
25. Indrzejczak, A.: Sequents and Trees. An Introduction to the Theory and Applications of Propositional Sequent Calculi. Birkhäuser (2021)
26. Indrzejczak, A.: Russellian definite description theory–a proof-theoretic approach. Rev. Symb. Logic **16**(2), 624–649 (2023)
27. Indrzejczak, A.: Bisequent calculus for four-valued quasi-relevant logics; cut elimination and interpolation. J. Autom. Reason, submitted
28. Indrzejczak, A., Zawidzki, M.: Tableaux for free logics with descriptions. In: Das, A., Negri, S. (eds.) TABLEAUX 2021. LNCS (LNAI), vol. 12842, pp. 56–73. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86059-2_4
29. Jaśkowski, S.: Recherches sur le système de la logique intuitioniste. Actes Congr. Int. phil. sci. **6**, 58–61 (1936)
30. Jaśkowski, S.: Rachunek zdań dla systemów dedukcyjnych sprzecznych. Studia Societatis Scientiarum Torunensis, Sectio A **I**(5), 57–77 (1948). English translation: A propositional calculus for inconsistent deductive systems. Logic Logical Philos. **7**, 35–56 (1999)
31. Kleene, S.C.: On a notation for ordinal numbers. J. Symb. Logic **3**(1), 150–155 (1938)
32. Komendantskaya, E.Y.: Functional expressibility of regular Kleene's logics. Logical Invest. **15**, 116–128 (2009). (in Russian)
33. Lehmann, S.: Strict Fregean free logic. J. Philos. Log. **23**(3), 307–336 (1994)
34. Łukasiewicz, J.: On three-valued logic (English translation of Łukasiewicz's paper of 1920). In: Borkowski, L. (ed.) Jan Łukasiewicz: Selected Works, pp. 87–88. North-Holland Publishing Company (1970)
35. Marcos, J.: On a problem of da Costa. In: Sica, G. (ed.) Essays of the Foundations of Mathematics and Logic, vol. 2, pp. 53–69. Polimetrica (2005)
36. McCarty, J.: A basis for a mathematical theory of computation. Stud. Logic Found. Math. **35**, 33–70 (1963)
37. Metcalfe, G., Olivetti, N., Gabbay, D.: Proof Theory for Fuzzy Logics. Springer, Cham (2008)
38. Osorio, M., Carballido, J.L.: Brief study of $\mathbf{G_3'}$ logic. J. Appl. Non-Classical Logic **18**(4), 475–499 (2008)
39. Paoli, F., Pra Baldi, M.: Proof theory of paraconsistent weak Kleene logic. Stud. Logica **108**(4), 779–802 (2020)
40. Petrukhin, Y.: Natural deduction for Post's logics and their duals. Log. Univers. **12**(1–2), 83–100 (2018)
41. Post, E.: Introduction to a general theory of elementary propositions. Am. J. Math. **43**, 163–185 (1921)
42. Priest, G.: The logic of paradox. J. Philos. Log. **8**(1), 219–241 (1979)
43. Rescher, N.: Many-Valued Logic. McGraw Hill, New York (1969)
44. Rozonoer, L.: On interpretation of inconsistent theories. Inf. Sci. **47**(3), 243–266 (1989)
45. Rousseau, G.: Sequents in many valued logic. Fundamenta Mathematicae **LX**(1), 22–23 (1967)
46. Sette, A.M., Carnieli, W.A.: Maximal weakly-intuitionistic logics. Stud. Logica **55**(1), 181–203 (1995)

47. Sette, A.M.: On propositional calculus P_1. Mathematica Japonica **18**(3), 173–180 (1973)
48. Słupecki, J.: Proof of axiomatizability of full many-valued systems of calculus of propositions. Stud. Logica **29**, 155–168 (1971). [English translation of the paper from 1939]
49. Słupecki, E., Bryll, J., Prucnal, T.: Some remarks on the three-valued logic of J. Łukasiewicz. Stud. Logica **21**(1), 45–70 (1967)
50. Sobociński, B.: Axiomatization of certain many-valued systems of the theory of deduction. Roczniki Prac Naukowych Zrzeszenia Asystentów Uniwersytetu Józefa Piłsudskiego w Warszawie **1**, 399–419 (1936)
51. Sobociński, B.: Axiomatization of a partial system of three-valued calculus of propositions. J. Comput. Syst. **1**, 23–55 (1952)
52. Stenlund, S.: The logic of description and existence. Filosofiska Studier **18**. Uppsala (1973)
53. Suchoń, W.: La methode de Smullyan de construire le calcul n-valent de Łukasiewicz avec implication and negation. Rep. Math. Logic **2**, 37–42 (1974)
54. Suszko, R.: The Fregean axiom and Polish mathematical logic in the 1920's. Stud. Logica **36**(4), 373–380 (1977)
55. Surma, S.J.: A method of the construction of finite Łukasiewiczian algebras and its application to a Gentzen-style characterisation of finite logics. Rep. Math. Logic **2**, 49–54 (1974)
56. Takahashi, M.: Many-valued logics of extended Gentzen style I. Sci. Rep. Tokyo Kyoiku Daigaku **9**(231), 95–116 (1967)
57. Tomova, N.E.: A lattice of implicative extensions of regular Kleene's logics. Rep. Math. Logic **47**, 173–182 (2012)
58. Wintein, S., Muskens, R.: Interpolation methods for Dunn logics and their extensions. Stud. Logica **105**(6), 1319–1347 (2017). https://doi.org/10.1007/s11225-017-9720-5

Proving Almost-Sure Innermost Termination of Probabilistic Term Rewriting Using Dependency Pairs

Jan-Christoph Kassing(✉) and Jürgen Giesl(✉)

LuFG Informatik 2, RWTH Aachen University, Aachen, Germany
kassing@cs.rwth-aachen.de, giesl@informatik.rwth-aachen.de

Abstract. Dependency pairs are one of the most powerful techniques to analyze termination of term rewrite systems (TRSs) automatically. We adapt the dependency pair framework to the probabilistic setting in order to prove almost-sure innermost termination of probabilistic TRSs. To evaluate its power, we implemented the new framework in our tool AProVE.

1 Introduction

Techniques and tools to analyze innermost termination of term rewrite systems (TRSs) automatically are successfully used for termination analysis of programs in many languages (e.g., Java [10,35,38], Haskell [18], and Prolog [19]). While there exist several classical orderings for proving termination of TRSs (e.g., based on polynomial interpretations [30]), a *direct* application of these orderings is usually too weak for TRSs that result from actual programs. However, these orderings can be used successfully within the *dependency pair* (DP) framework [2,16,17]. This framework allows for modular termination proofs (e.g., which apply different orderings in different sub-proofs) and is one of the most powerful techniques for termination analysis of TRSs that is used in essentially all current termination tools for TRSs, e.g., AProVE [20], MU-TERM [22], NaTT [40], TTT2 [29], etc.

On the other hand, *probabilistic* programs are used to describe randomized algorithms and probability distributions, with applications in many areas. To use TRSs also for such programs, *probabilistic term rewrite systems* (PTRSs) were introduced in [8,9]. In the probabilistic setting, there are several notions of "termination". A program is *almost-surely terminating* (AST) if the probability for termination is 1. As remarked in [24]: "AST is the classical and most widely-studied problem that extends termination of non-probabilistic programs, and is considered as a core problem in the programming languages community". A strictly stronger notion is *positive almost-sure termination* (PAST), which

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2) and DFG Research Training Group 2236 UnRAVeL.

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 344–364, 2023.
https://doi.org/10.1007/978-3-031-38499-8_20

requires that the expected runtime is finite. While there exist many automatic approaches to prove (P)AST of imperative programs on numbers (e.g., [1,4,11, 15,21,24–26,32–34,36]), there are only few automatic approaches for programs with complex non-tail recursive structure [7,12], and even less approaches which are also suitable for algorithms on recursive data structures [3,6,31,39]. The approach of [39] focuses on algorithms on lists and [31] mainly targets algorithms on trees, but they cannot easily be adjusted to other (possibly user-defined) data structures. The calculus of [6] considers imperative programs with stack, heap, and pointers, but it is not yet automated. Moreover, the approaches of [3,6,31,39] analyze expected runtime, while we focus on AST.

PTRSs can be used to model algorithms (possibly with complex recursive structure) operating on algebraic data types. While PTRSs were introduced in [8, 9], the first (and up to now only) tool to analyze their termination automatically was presented in [3], where orderings based on interpretations were adapted to prove PAST. Moreover, [14] extended general concepts of abstract rewrite systems (e.g., confluence and uniqueness of normal forms) to the probabilistic setting.

As mentioned, already for non-probabilistic TRSs a *direct* application of orderings (as in [3]) is limited in power. To obtain a powerful approach, one should combine such orderings in a modular way, as in the DP framework. In this paper, we show for the first time that an adaption of dependency pairs to the probabilistic setting is possible and present the first DP framework for probabilistic term rewriting. Since the crucial idea of dependency pairs is the modularization of the termination proof, we analyze AST instead of PAST, because it is well known that AST is compositional, while PAST is not (see, e.g., [25]). We also present a novel adaption of the technique from [3] for the direct application of polynomial interpretations in order to prove AST (instead of PAST) of PTRSs.

We start by briefly recapitulating the DP framework for non-probabilistic TRSs in Sect. 2. Then we recall the definition of PTRSs based on [3,9,14] in Sect. 3 and introduce a novel way to prove AST using polynomial interpretations automatically. In Sect. 4 we present our new probabilistic DP framework. The implementation of our approach in the tool AProVE is evaluated in Sect. 5. We refer to [28] for all proofs (which are much more involved than the original proofs for the non-probabilistic DP framework from [2,16,17]).

2 The DP Framework

We assume familiarity with term rewriting [5] and regard TRSs over a finite signature Σ and a set of variables $\mathcal{V}$. A *polynomial interpretation* Pol is a Σ-algebra with carrier set $\mathbb{N}$ which maps every function symbol $f \in \Sigma$ to a polynomial $f_{\mathrm{Pol}} \in \mathbb{N}[\mathcal{V}]$. For a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, $\mathrm{Pol}(t)$ denotes the interpretation of t by the Σ-algebra Pol. An arithmetic inequation like $\mathrm{Pol}(t_1) > \mathrm{Pol}(t_2)$ *holds* if it is true for all instantiations of its variables by natural numbers.

Theorem 1 (Termination With Polynomial Interpretations [30]**).** *Let $\mathcal{R}$ be a TRS and let* $\mathrm{Pol} : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathbb{N}[\mathcal{V}]$ *be a monotonic polynomial interpretation (i.e., $x > y$ implies $f_{\mathrm{Pol}}(\dots, x, \dots) > f_{\mathrm{Pol}}(\dots, y, \dots)$ for all $f \in \Sigma$). If for every $\ell \to r \in \mathcal{R}$, we have* $\mathrm{Pol}(\ell) > \mathrm{Pol}(r)$*, then $\mathcal{R}$ is terminating.*

The search for polynomial interpretations is usually automated by SMT solving. Instead of polynomials over the naturals, Theorem 1 (and the other termination criteria in the paper) can also be extended to polynomials over the non-negative reals, by requiring that whenever a term is "strictly decreasing", then its interpretation decreases at least by a certain fixed amount $\delta > 0$.

Example 2. Consider the TRS $\mathcal{R}_{\mathsf{div}} = \{(1), \dots, (4)\}$ for division from [2].

$$\mathsf{minus}(x, \mathcal{O}) \to x \qquad (1) \qquad \mathsf{div}(\mathcal{O}, \mathsf{s}(y)) \to \mathcal{O} \qquad (3)$$

$$\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{minus}(x, y) \qquad (2) \qquad \mathsf{div}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{s}\left(\mathsf{div}(\mathsf{minus}(x, y), \mathsf{s}(y))\right) \qquad (4)$$

Termination of $\mathcal{R}_{\mathsf{minus}} = \{(1), (2)\}$ can be proved by the polynomial interpretation that maps $\mathsf{minus}(x, y)$ to $x + y + 1$, $\mathsf{s}(x)$ to $x + 1$, and $\mathcal{O}$ to 0. However, a direct application of classical techniques like polynomial interpretations fails for $\mathcal{R}_{\mathsf{div}}$. These techniques correspond to so-called (quasi-)simplification orderings [13] which cannot handle rules like (4) where the right-hand side is embedded in the left-hand side if y is instantiated with $\mathsf{s}(x)$. In contrast, the dependency pair framework is able to prove termination of $\mathcal{R}_{\mathsf{div}}$ automatically.

We now recapitulate the DP framework and its core processors, and refer to, e.g., [2,16,17,23] for more details. In this paper, we restrict ourselves to the DP framework for *innermost* rewriting (denoted "$\xrightarrow{\mathsf{i}}_{\mathcal{R}}$"), because our adaption to the probabilistic setting relies on this evaluation strategy (see Sect. 4.1).

Definition 3 (Dependency Pair). *Let $\mathcal{R}$ be a (finite) TRS. We decompose its signature $\Sigma = \Sigma_C \uplus \Sigma_D$ such that $f \in \Sigma_D$ if $f =$* $\mathrm{root}(\ell)$ *for some rule $\ell \to r \in \mathcal{R}$. The symbols in Σ_C and Σ_D are called* constructors *and* defined symbols*, respectively. For every $f \in \Sigma_D$, we introduce a fresh* tuple symbol *$f^{\#}$ of the same arity. Let $\Sigma^{\#}$ denote the set of all tuple symbols. To ease readability, we often write F instead of $\mathsf{f}^{\#}$. For any term $t = f(t_1, \dots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$ with $f \in \Sigma_D$, let $t^{\#} = f^{\#}(t_1, \dots, t_n)$. Moreover, for any $r \in \mathcal{T}(\Sigma, \mathcal{V})$, let* $\mathrm{Sub}_D(r)$ *be the set of all subterms of r with defined root symbol. For a rule $\ell \to r$ with* $\mathrm{Sub}_D(r) = \{t_1, \dots, t_n\}$*, one obtains the n dependency pairs (DPs) $\ell^{\#} \to t_i^{\#}$ with $1 \le i \le n$. $\mathcal{DP}(\mathcal{R})$ denotes the set of all dependency pairs of $\mathcal{R}$.*

Example 4. For the TRS $\mathcal{R}_{\mathsf{div}}$ from Example 2, we get the following dependency pairs.

$$\mathsf{M}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{M}(x, y) \quad (5) \qquad \mathsf{D}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{M}(x, y) \qquad (6)$$

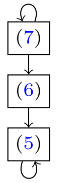
$$\mathsf{D}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{D}(\mathsf{minus}(x, y), \mathsf{s}(y)) \qquad (7)$$

The DP framework uses *DP problems* $(\mathcal{D}, \mathcal{R})$ where $\mathcal{D}$ is a (finite) set of DPs and $\mathcal{R}$ is a (finite) TRS. A (possibly infinite) sequence $t_0^{\#}, t_1^{\#}, t_2^{\#}, \dots$ with $t_i^{\#} \xrightarrow{\mathsf{i}}_{\mathcal{D},\mathcal{R}} \circ \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} t_{i+1}^{\#}$ for all i is an (innermost) $(\mathcal{D}, \mathcal{R})$-*chain*. Here, $\xrightarrow{\mathsf{i}}_{\mathcal{D},\mathcal{R}}$ is the restriction of $\to_{\mathcal{D}}$ to rewrite steps where the used redex is in normal form w.r.t. $\mathcal{R}$. A chain represents subsequent "function calls" in evaluations. Between two function calls (corresponding to steps with $\mathcal{D}$) one can evaluate the arguments with $\mathcal{R}$. For example, $\mathsf{D}(\mathsf{s}^2(\mathcal{O}), \mathsf{s}(\mathcal{O}))$, $\mathsf{D}(\mathsf{s}(\mathcal{O}), \mathsf{s}(\mathcal{O}))$ is a $(\mathcal{DP}(\mathcal{R}_{\mathsf{div}}), \mathcal{R}_{\mathsf{div}})$-chain, as $\mathsf{D}(\mathsf{s}^2(\mathcal{O}), \mathsf{s}(\mathcal{O})) \xrightarrow{\mathsf{i}}_{\mathcal{DP}(\mathcal{R}_{\mathsf{div}}), \mathcal{R}_{\mathsf{div}}} \mathsf{D}(\mathsf{minus}(\mathsf{s}(\mathcal{O}), \mathcal{O}), \mathsf{s}(\mathcal{O})) \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}_{\mathsf{div}}} \mathsf{D}(\mathsf{s}(\mathcal{O}), \mathsf{s}(\mathcal{O}))$, where $\mathsf{s}^2(\mathcal{O})$ is $\mathsf{s}(\mathsf{s}(\mathcal{O}))$.

A DP problem $(\mathcal{D}, \mathcal{R})$ is called *innermost terminating* (iTerm) if there is no infinite innermost $(\mathcal{D}, \mathcal{R})$-chain. The main result on dependency pairs is the *chain criterion* which states that a TRS $\mathcal{R}$ is iTerm iff $(\mathcal{DP}(\mathcal{R}), \mathcal{R})$ is iTerm. The key idea of the DP framework is a *divide-and-conquer* approach which applies *DP processors* to transform DP problems into simpler sub-problems. A *DP processor* Proc has the form $\mathrm{Proc}(\mathcal{D}, \mathcal{R}) = \{(\mathcal{D}_1, \mathcal{R}_1), \ldots, (\mathcal{D}_n, \mathcal{R}_n)\}$, where $\mathcal{D}, \mathcal{D}_1, \ldots, \mathcal{D}_n$ are sets of dependency pairs and $\mathcal{R}, \mathcal{R}_1, \ldots, \mathcal{R}_n$ are TRSs. A processor Proc is *sound* if $(\mathcal{D}, \mathcal{R})$ is iTerm whenever $(\mathcal{D}_i, \mathcal{R}_i)$ is iTerm for all $1 \leq i \leq n$. It is *complete* if $(\mathcal{D}_i, \mathcal{R}_i)$ is iTerm for all $1 \leq i \leq n$ whenever $(\mathcal{D}, \mathcal{R})$ is iTerm.

So given a TRS $\mathcal{R}$, one starts with the initial DP problem $(\mathcal{DP}(\mathcal{R}), \mathcal{R})$ and applies sound (and preferably complete) DP processors repeatedly until all sub-problems are "solved" (i.e., sound processors transform them to the empty set). This allows for modular termination proofs, since different techniques can be applied on each resulting "sub-problem" $(\mathcal{D}_i, \mathcal{R}_i)$. The following three theorems recapitulate the three most important processors of the DP framework.

The (innermost) $(\mathcal{D}, \mathcal{R})$-*dependency graph* is a control flow graph that indicates which dependency pairs can be used after each other in a chain. Its node set is $\mathcal{D}$ and there is an edge from $\ell_1^{\#} \to t_1^{\#}$ to $\ell_2^{\#} \to t_2^{\#}$ if there exist substitutions σ_1, σ_2 such that $t_1^{\#}\sigma_1 \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} \ell_2^{\#}\sigma_2$, and both $\ell_1^{\#}\sigma_1$ and $\ell_2^{\#}\sigma_2$ are in normal form w.r.t. $\mathcal{R}$. Any infinite $(\mathcal{D}, \mathcal{R})$-chain corresponds to an infinite path in the dependency graph, and since the graph is finite, this infinite path must end in some strongly connected component (SCC).[1] Hence, it suffices to consider the SCCs of this graph independently. The $(\mathcal{DP}(\mathcal{R}_{\mathsf{div}}), \mathcal{R}_{\mathsf{div}})$-dependency graph can be seen on the right.



Theorem 5 (Dep. Graph Processor). *For the SCCs* $\mathcal{D}_1, \ldots, \mathcal{D}_n$ *of the* $(\mathcal{D}, \mathcal{R})$*-dependency graph,* $\mathrm{Proc}_{\mathsf{DG}}(\mathcal{D}, \mathcal{R}) = \{(\mathcal{D}_1, \mathcal{R}), \ldots, (\mathcal{D}_n, \mathcal{R})\}$ *is sound and complete.*

While the exact dependency graph is not computable in general, there are several techniques to over-approximate it automatically, see, e.g., [2,17,23]. In our example, applying $\mathrm{Proc}_{\mathsf{DG}}$ to the initial problem $(\mathcal{DP}(\mathcal{R}_{\mathsf{div}}), \mathcal{R}_{\mathsf{div}})$ results in the smaller problems $\big(\{(5)\}, \mathcal{R}_{\mathsf{div}}\big)$ and $\big(\{(7)\}, \mathcal{R}_{\mathsf{div}}\big)$ that can be treated separately.

The next processor removes rules that cannot be used to evaluate right-hand sides of dependency pairs when their variables are instantiated with normal forms.

[1] Here, a set $\mathcal{D}'$ of dependency pairs is an *SCC* if it is a maximal cycle, i.e., it is a maximal set such that for any $\ell_1^{\#} \to t_1^{\#}$ and $\ell_2^{\#} \to t_2^{\#}$ in $\mathcal{D}'$ there is a non-empty path from $\ell_1^{\#} \to t_1^{\#}$ to $\ell_2^{\#} \to t_2^{\#}$ which only traverses nodes from $\mathcal{D}'$.

Theorem 6 (Usable Rules Processor). *Let $\mathcal{R}$ be a TRS. For every $f \in \Sigma \uplus \Sigma^{\#}$ let* $\mathrm{Rules}_{\mathcal{R}}(f) = \{\ell \to r \in \mathcal{R} \mid \mathrm{root}(\ell) = f\}$*. For any $t \in \mathcal{T}\left(\Sigma \uplus \Sigma^{\#}, \mathcal{V}\right)$, its* usable rules $\mathcal{U}_{\mathcal{R}}(t)$ *are the smallest set such that $\mathcal{U}_{\mathcal{R}}(x) = \varnothing$ for all $x \in \mathcal{V}$ and* $\mathcal{U}_{\mathcal{R}}(f(t_1, \dots, t_n)) = \mathrm{Rules}_{\mathcal{R}}(f) \cup \bigcup_{i=1}^{n} \mathcal{U}_{\mathcal{R}}(t_i) \cup \bigcup_{\ell \to r \in \mathrm{Rules}_{\mathcal{R}}(f)} \mathcal{U}_{\mathcal{R}}(r)$*. The* usable rules *for the DP problem $(\mathcal{D}, \mathcal{R})$ are $\mathcal{U}(\mathcal{D}, \mathcal{R}) = \bigcup_{\ell^{\#} \to t^{\#} \in \mathcal{D}} \mathcal{U}_{\mathcal{R}}(t^{\#})$. Then* $\mathrm{Proc}_{\mathtt{UR}}(\mathcal{D}, \mathcal{R}) = \{(\mathcal{D}, \mathcal{U}(\mathcal{D}, \mathcal{R}))\}$ *is sound but not complete.*[2]

For the DP problem $\left(\{(7)\}, \mathcal{R}_{\mathsf{div}}\right)$ only the minus-rules are usable and thus $\mathrm{Proc}_{\mathtt{UR}}\left(\{(7)\}, \mathcal{R}_{\mathsf{div}}\right) = \left\{\left(\{(7)\}, \{(1), (2)\}\right)\right\}$. For $\left(\{(5)\}, \mathcal{R}_{\mathsf{div}}\right)$ there are no usable rules at all, and thus $\mathrm{Proc}_{\mathtt{UR}}\left(\{(5)\}, \mathcal{R}_{\mathsf{div}}\right) = \left\{\left(\{(5)\}, \varnothing\right)\right\}$.

The last processor adapts classical orderings like polynomial interpretations to DP problems.[3] In contrast to their direct application in Theorem 1, we may now use weakly monotonic polynomials f_{Pol} that do not have to depend on all of their arguments. The reduction pair processor requires that all rules and dependency pairs are weakly decreasing and it removes those DPs that are strictly decreasing.

Theorem 7 (Reduction Pair Processor with Polynomial Interpretations). *Let* $\mathrm{Pol} : \mathcal{T}\left(\Sigma \uplus \Sigma^{\#}, \mathcal{V}\right) \to \mathbb{N}[\mathcal{V}]$ *be a weakly monotonic polynomial interpretation (i.e., $x \geq y$ implies $f_{\mathrm{Pol}}(\dots, x, \dots) \geq f_{\mathrm{Pol}}(\dots, y, \dots)$ for all $f \in \Sigma \uplus \Sigma^{\#}$). Let $\mathcal{D} = \mathcal{D}_{\geq} \uplus \mathcal{D}_{>}$ with $\mathcal{D}_{>} \neq \varnothing$ such that:*

(1) For every $\ell \to r \in \mathcal{R}$, we have $\mathrm{Pol}(\ell) \geq \mathrm{Pol}(r)$*.*
(2) For every $\ell^{\#} \to t^{\#} \in \mathcal{D}$, we have $\mathrm{Pol}(\ell^{\#}) \geq \mathrm{Pol}(t^{\#})$*.*
(3) For every $\ell^{\#} \to t^{\#} \in \mathcal{D}_{>}$, we have $\mathrm{Pol}(\ell^{\#}) > \mathrm{Pol}(t^{\#})$*.*

Then $\mathrm{Proc}_{\mathtt{RP}}(\mathcal{D}, \mathcal{R}) = \{(\mathcal{D}_{\geq}, \mathcal{R})\}$ *is sound and complete.*

The constraints of the reduction pair processor for the remaining DP problems $(\{(7)\}, \{(1), (2)\})$ and $(\{(5)\}, \varnothing)$ are satisfied by the polynomial interpretation which maps $\mathcal{O}$ to 0, $\mathsf{s}(x)$ to $x + 1$, and all other non-constant function symbols to the projection on their first arguments. Since (7) and (5) are strictly decreasing, $\mathrm{Proc}_{\mathtt{RP}}$ transforms both $(\{(7)\}, \{(1), (2)\})$ and $(\{(5)\}, \varnothing)$ into DP problems of the form $(\varnothing, \dots)$. As $\mathrm{Proc}_{\mathtt{DG}}(\varnothing, \dots) = \varnothing$ and all processors used are sound, this means that there is no infinite innermost chain for the initial DP problem $(\mathcal{DP}(\mathcal{R}_{\mathsf{div}}), \mathcal{R}_{\mathsf{div}})$ and thus, $\mathcal{R}_{\mathsf{div}}$ is innermost terminating.

3 Probabilistic Term Rewriting

Now we recapitulate *probabilistic TRSs* [3,9,14] and present a novel criterion to prove almost-sure termination automatically by adapting the direct application

[2] For a complete version of the usable rules processor, one has to use a more involved notion of DP problems with more components that we omit here for readability [16].

[3] In this paper, we only regard the reduction pair processor with polynomial interpretations, because for most other classical orderings it is not clear how to extend them to probabilistic TRSs, where one has to consider "expected values of terms".

of polynomial interpretations from Theorem 1 to PTRSs. In contrast to TRSs, a PTRS has finite[4] multi-distributions on the right-hand side of rewrite rules.

Definition 8 (Multi-Distribution). *A finite* multi-distribution *μ on a set $A \neq \varnothing$ is a finite multiset of pairs $(p : a)$, where $0 < p \leq 1$ is a probability and $a \in A$, such that $\sum_{(p:a)\in\mu} p = 1$.* $\mathrm{FDist}(A)$ *is the set of all finite multi-distributions on A. For $\mu \in \mathrm{FDist}(A)$, its* support *is the multiset* $\mathrm{Supp}(\mu) = \{a \mid (p : a) \in \mu \text{ for some } p\}$.

Definition 9 (PTRS). *A* probabilistic rewrite rule *is a pair $\ell \to \mu \in \mathcal{T}(\Sigma,\mathcal{V}) \times \mathrm{FDist}(\mathcal{T}(\Sigma,\mathcal{V}))$ such that $\ell \notin \mathcal{V}$ and $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$ for every $r \in \mathrm{Supp}(\mu)$. A* probabilistic TRS *(PTRS) is a finite set $\mathcal{R}$ of probabilistic rewrite rules. Similar to TRSs, the PTRS $\mathcal{R}$ induces a* rewrite relation *$\to_{\mathcal{R}} \subseteq \mathcal{T}(\Sigma,\mathcal{V}) \times \mathrm{FDist}(\mathcal{T}(\Sigma,\mathcal{V}))$ where $s \to_{\mathcal{R}} \{p_1 : t_1, \ldots, p_k : t_k\}$ if there is a position π, a rule $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\} \in \mathcal{R}$, and a substitution σ such that $s|_\pi = \ell\sigma$ and $t_j = s[r_j\sigma]_\pi$ for all $1 \leq j \leq k$. We call $s \to_{\mathcal{R}} \mu$ an* innermost *rewrite step (denoted $s \overset{\mathsf{i}}{\to}_{\mathcal{R}} \mu$) if every proper subterm of the used redex $\ell\sigma$ is in normal form w.r.t. $\mathcal{R}$.*

Example 10. As an example, consider the PTRS $\mathcal{R}_{\mathsf{rw}}$ with the only rule $\mathsf{g}(x) \to \{1/2 : x,\ 1/2 : \mathsf{g}(\mathsf{g}(x))\}$, which corresponds to a symmetric random walk.

As proposed in [3], we *lift* $\to_{\mathcal{R}}$ to a rewrite relation between multi-distributions in order to track all probabilistic rewrite sequences (up to non-determinism) at once. For any $0 < p \leq 1$ and any $\mu \in \mathrm{FDist}(A)$, let $p \cdot \mu = \{(p \cdot q : a) \mid (q : a) \in \mu\}$.

Definition 11 (Lifting). *The* lifting *$\rightrightarrows \subseteq \mathrm{FDist}(\mathcal{T}(\Sigma,\mathcal{V})) \times \mathrm{FDist}(\mathcal{T}(\Sigma,\mathcal{V}))$ of a relation $\to \subseteq \mathcal{T}(\Sigma,\mathcal{V}) \times \mathrm{FDist}(\mathcal{T}(\Sigma,\mathcal{V}))$ is the smallest relation with:*

- *If $t \in \mathcal{T}(\Sigma,\mathcal{V})$ is in normal form w.r.t. $\to$, then $\{1 : t\} \rightrightarrows \{1 : t\}$.*
- *If $t \to \mu$, then $\{1 : t\} \rightrightarrows \mu$.*
- *If for all $1 \leq j \leq k$ there are $\mu_j, \nu_j \in \mathrm{FDist}(\mathcal{T}(\Sigma,\mathcal{V}))$ with $\mu_j \rightrightarrows \nu_j$ and $0 < p_j \leq 1$ with $\sum_{1\leq j\leq k} p_j = 1$, then $\bigcup_{1\leq j\leq k} p_j \cdot \mu_j \rightrightarrows \bigcup_{1\leq j\leq k} p_j \cdot \nu_j$.*

For a PTRS $\mathcal{R}$, we write $\rightrightarrows_{\mathcal{R}}$ and $\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{R}}$ for the liftings of $\to_{\mathcal{R}}$ and $\overset{\mathsf{i}}{\to}_{\mathcal{R}}$, respectively.

Example 12. For instance, we obtain the following $\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{R}_{\mathsf{rw}}}$-rewrite sequence:

$$\begin{aligned}\{1 : \mathsf{g}(\mathcal{O})\} &\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{R}_{\mathsf{rw}}} \{1/2 : \mathcal{O}, 1/2 : \mathsf{g}^2(\mathcal{O})\} \overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{R}_{\mathsf{rw}}} \{1/2 : \mathcal{O}, 1/4 : \mathsf{g}(\mathcal{O}), 1/4 : \mathsf{g}^3(\mathcal{O})\}\\ &\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{R}_{\mathsf{rw}}} \{1/2 : \mathcal{O}, 1/8 : \mathcal{O}, 1/8 : \mathsf{g}^2(\mathcal{O}), 1/8 : \mathsf{g}^2(\mathcal{O}), 1/8 : \mathsf{g}^4(\mathcal{O})\}\end{aligned}$$

Note that the two occurrences of $\mathcal{O}$ and $\mathsf{g}^2(\mathcal{O})$ in the multi-distribution above could be rewritten differently if the PTRS had rules resulting in different terms. So it should be distinguished from $\{5/8 : \mathcal{O}, 1/4 : \mathsf{g}^2(\mathcal{O}), 1/8 : \mathsf{g}^4(\mathcal{O})\}$.

[4] Since our goal is the automation of termination analysis, in this paper we restrict ourselves to finite PTRSs with finite multi-distributions.

To express the concept of almost-sure termination, one has to determine the probability for normal forms in a multi-distribution.

Definition 13 **($|\mu|_{\mathcal{R}}$).** *For a PTRS $\mathcal{R}$, $\mathtt{NF}_{\mathcal{R}} \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ denotes the set of all normal forms w.r.t. $\mathcal{R}$. For any $\mu \in \mathrm{FDist}(\mathcal{T}(\Sigma, \mathcal{V}))$, let $|\mu|_{\mathcal{R}} = \sum_{(p:t)\in\mu, t\in\mathtt{NF}_{\mathcal{R}}} p$.*

Example 14. Consider the multi-distribution $\{1/2 : \mathcal{O}, 1/8 : \mathcal{O}, 1/8 : \mathsf{g}^2(\mathcal{O}), 1/8 : \mathsf{g}^2(\mathcal{O}), 1/8 : \mathsf{g}^4(\mathcal{O})\}$ from Example 12 and $\mathcal{R}_{\mathsf{rw}}$ from Example 10. Then $|\mu|_{\mathcal{R}_{\mathsf{rw}}} = 1/2 + 1/8 = 5/8$.

Definition 15 **((Innermost) AST).** *Let $\mathcal{R}$ be a PTRS and $(\mu_n)_{n\in\mathbb{N}}$ be an infinite $\rightrightarrows_{\mathcal{R}}$-rewrite sequence, i.e., $\mu_n \rightrightarrows_{\mathcal{R}} \mu_{n+1}$ for all $n \in \mathbb{N}$. Note that $\lim\limits_{n\to\infty} |\mu_n|_{\mathcal{R}}$ exists, since $|\mu_n|_{\mathcal{R}} \leq |\mu_{n+1}|_{\mathcal{R}} \leq 1$ for all $n \in \mathbb{N}$. $\mathcal{R}$ is* almost-surely terminating (AST) *(*innermost almost-surely terminating (iAST)*) if $\lim\limits_{n\to\infty} |\mu_n|_{\mathcal{R}} = 1$ holds for every infinite $\rightrightarrows_{\mathcal{R}}$-rewrite sequence ($\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{R}}$-rewrite sequence) $(\mu_n)_{n\in\mathbb{N}}$.*

Example 16. For the (unique) infinite extension of the $\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{R}_{\mathsf{rw}}}$-rewrite sequence $(\mu_n)_{n\in\mathbb{N}}$ in Example 12, we have $\lim\limits_{n\to\infty} |\mu_n|_{\mathcal{R}} = 1$. Indeed, $\mathcal{R}_{\mathsf{rw}}$ is AST (but not PAST, i.e., the expected number of rewrite steps is infinite for every term containing g).

Theorem 17 introduces a novel technique to prove AST automatically using a direct application of polynomial interpretations.

Theorem 17 **(Proving AST with Polynomial Interpretations).** *Let $\mathcal{R}$ be a PTRS, let $\mathrm{Pol} : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathbb{N}[\mathcal{V}]$ be a monotonic, multilinear[5] polynomial interpretation (i.e., for all $f \in \Sigma$, all monomials of $f_{\mathrm{Pol}}(x_1, \ldots, x_n)$ have the form $c \cdot x_1^{e_1} \cdot \ldots \cdot x_n^{e_n}$ with $c \in \mathbb{N}$ and $e_1, \ldots, e_n \in \{0, 1\}$). If for every rule $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\} \in \mathcal{R}$,*

(1) there exists a $1 \leq j \leq k$ with $\mathrm{Pol}(\ell) > \mathrm{Pol}(r_j)$ and
(2) $\mathrm{Pol}(\ell) \geq \sum_{1\leq j\leq k} p_j \cdot \mathrm{Pol}(r_j)$,

then $\mathcal{R}$ is AST.

In [3], it was shown that PAST can be proved by using multilinear polynomials and requiring a strict decrease in the expected value of each rule. In contrast, we only require a weak decrease of the expected value in (2) and in addition, at least one term in the support of the right-hand side must become strictly smaller (1). As mentioned, the proof for Theorem 17 (and for all our other new results and observations) can be found in [28]. The proof idea is based on [32], but it extends their approach from while-programs on integers to terms. However, in contrast to [32], PTRSs can only deal with constant probabilities, since all variables stand for terms, not for numbers. Note that the constraints (1) and (2) of our new criterion in Theorem 17 are equivalent to the constraint of the classical Theorem 1 in the special case where the PTRS is in fact a TRS (i.e., all rules have the form $\ell \to \{1 : r\}$).

[5] As in [3], multilinearity ensures "monotonicity" w.r.t. expected values, since multilinearity implies $f_{\mathrm{Pol}}(\ldots, \sum_{1\leq j\leq k} p_j \cdot \mathrm{Pol}(r_j), \ldots) = \sum_{1\leq j\leq k} p_j \cdot \mathrm{Pol}(f(\ldots, r_j, \ldots))$.

Example 18. To prove that $\mathcal{R}_{\mathsf{rw}}$ is AST with Theorem 17, we can use the polynomial interpretation that maps $\mathsf{g}(x)$ to $x+1$ and $\mathcal{O}$ to 0.

4 Probabilistic Dependency Pairs

We introduce our new adaption of DPs to the probabilistic setting in Sect. 4.1. Then we present the processors for the probabilistic DP framework in Sect. 4.2.

4.1 Dependency Tuples and Chains for Probabilistic Term Rewriting

We first show why straightforward adaptions are unsound. A natural idea to define DPs for probabilistic rules $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\} \in \mathcal{R}$ would be (8) or (9):

$$\{\, \ell^{\#} \to \{p_1 : r_1, \ldots, p_i : t_j^{\#}, \ldots, p_k : r_k\} \mid t_j \in \mathrm{Sub}_D(r_j) \text{ with } 1 \leq j \leq k \,\} \quad (8)$$

$$\{\, \ell^{\#} \to \{p_1 : t_1^{\#}, \ldots, p_k : t_k^{\#}\} \mid t_j \in \mathrm{Sub}_D(r_j) \text{ for all } 1 \leq j \leq k \,\} \quad (9)$$

For (9), if $\mathrm{Sub}_D(r_j) = \varnothing$, then we insert a fresh constructor $\bot$ into $\mathrm{Sub}_D(r_j)$ that does not occur in $\mathcal{R}$. So in both (8) and (9), we replace r_j by a single term $t_j^{\#}$ in the right-hand side. The following example shows that this notion of probabilistic DPs does not yield a sound chain criterion. Consider the PTRSs $\mathcal{R}_1$ and $\mathcal{R}_2$:

$$\mathcal{R}_1 = \{\mathsf{g} \to \{1/2 : \mathcal{O}, 1/2 : \mathsf{f}(\mathsf{g}, \mathsf{g})\}\} \quad \mathcal{R}_2 = \{\mathsf{g} \to \{1/2 : \mathcal{O}, 1/2 : \mathsf{f}(\mathsf{g}, \mathsf{g}, \mathsf{g})\}\} \quad (10)$$

$\mathcal{R}_1$ is AST since it corresponds to a symmetric random walk stopping at 0, where the number of gs denotes the current position. In contrast, $\mathcal{R}_2$ is not AST as it corresponds to a random walk where there is an equal chance of reducing the number of gs by 1 or increasing it by 2. For both $\mathcal{R}_1$ and $\mathcal{R}_2$, (8) and (9) would result in the only dependency pair $\mathsf{G} \to \{1/2 : \mathcal{O}, 1/2 : \mathsf{G}\}$ and $\mathsf{G} \to \{1/2 : \bot, 1/2 : \mathsf{G}\}$, resp. Rewriting with this DP is clearly AST, since it corresponds to a program that flips a coin until one gets head and then terminates. So the definitions (8) and (9) would not yield a sound approach for proving AST.

$\mathcal{R}_1$ and $\mathcal{R}_2$ show that the number of occurrences of the same subterm in the right-hand side r of a rule matters for AST. Thus, we now regard the *multiset* $\mathrm{MSub}_D(r)$ of all subterms of r with defined root symbol to ensure that multiple occurrences of the same subterm in r are taken into account. Moreover, instead of pairs we regard *dependency tuples* which consider all subterms with defined root in r at once. Dependency tuples were already used when adapting DPs for complexity analysis of (non-probabilistic) TRSs [37]. We now adapt them to the probabilistic setting and present a novel rewrite relation for dependency tuples.

Definition 19 (Transformation *dp***).** *If* $\mathrm{MSub}_D(r) = \{t_1, \ldots, t_n\}$, *then we define* $dp(r) = \mathsf{c}_n(t_1^{\#}, \ldots, t_n^{\#})$. *To make* $dp(r)$ *unique, we use the lexicographic ordering* $<$ *on positions where* $t_i = r|_{\pi_i}$ *and* $\pi_1 < \ldots < \pi_n$. *Here, we extend* Σ_C *by fresh* compound *constructor symbols* c_n *of arity* n *for* $n \in \mathbb{N}$.

When rewriting a subterm $t_i^\#$ of $\mathsf{c}_n(t_1^\#, \dots, t_n^\#)$ with a dependency tuple, one obtains terms with nested compound symbols. To abstract from nested compound symbols and from the order of their arguments, we introduce the following normalization.

Definition 20 (Normalizing Compound Terms). *For any term t, its* content *$cont(t)$ is the multiset defined by $cont(\mathsf{c}_n(t_1, \dots, t_n)) = cont(t_1) \cup \dots \cup cont(t_n)$ and $cont(t) = \{t\}$ otherwise. For any term t with $cont(t) = \{t_1, \dots, t_n\}$, the term $\mathsf{c}_n(t_1, \dots, t_n)$ is a* normalization *of t. For two terms t, t', we define $t \approx t'$ if $cont(t) = cont(t')$. We define $\approx$ on multi-distributions in a similar way: whenever $t_j \approx t_j'$ for all $1 \leq j \leq k$, then $\{p_1 : t_1, \dots, p_k : t_k\} \approx \{p_1 : t_1', \dots, p_k : t_k'\}$.*

So for example, $\mathsf{c}_3(x, x, y)$ is a normalization of $\mathsf{c}_2(\mathsf{c}_1(x), \mathsf{c}_2(x, y))$. We do not distinguish between terms and multi-distributions that are equal w.r.t. $\approx$ and we write $\mathsf{c}_n(t_1, \dots, t_n)$ for any term t with a compound root symbol where $cont(t) = \{t_1, \dots, t_n\}$, i.e., we consider all such t to be normalized.

For any rule $\ell \to \{p_1 : r_1, \dots, p_k : r_k\} \in \mathcal{R}$, the natural idea would be to define its *dependency tuple (DT)* as $\ell^\# \to \{p_1 : dp(r_1), \dots, p_k : dp(r_k)\}$. Then innermost *chains* in the probabilistic setting would result from alternating a DT-step with an arbitrary number of $\mathcal{R}$-steps (using $\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{R}}^*$). However, such chains would not necessarily correspond to the original rewrite sequence and thus, the resulting chain criterion would not be sound.

Example 21. Consider the PTRS $\mathcal{R}_3 = \{\mathsf{f}(\mathcal{O}) \to \{1 : \mathsf{f}(\mathsf{a})\}, \mathsf{a} \to \{1/2 : \mathsf{b}_1, 1/2 : \mathsf{b}_2\}, \mathsf{b}_1 \to \{1 : \mathcal{O}\}, \mathsf{b}_2 \to \{1 : \mathsf{f}(\mathsf{a})\}\}$. Its DTs would be $\mathcal{D}_3 = \{\mathsf{F}(\mathcal{O}) \to \{1 : \mathsf{c}_2(\mathsf{F}(\mathsf{a}), \mathsf{A})\}, \mathsf{A} \to \{1/2 : \mathsf{c}_1(\mathsf{B}_1), 1/2 : \mathsf{c}_1(\mathsf{B}_2)\}, \mathsf{B}_1 \to \{1 : \mathsf{c}_0\}, \mathsf{B}_2 \to \{1 : \mathsf{c}_2(\mathsf{F}(\mathsf{a}), \mathsf{A})\}\}$. $\mathcal{R}_3$ is not iAST, because one can extend the rewrite sequence

$$\{1{:}\mathsf{f}(\mathcal{O})\} \overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{R}_3} \{1{:}\mathsf{f}(\mathsf{a})\} \overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{R}_3} \{1/2{:}\mathsf{f}(\mathsf{b}_1), 1/2{:}\mathsf{f}(\mathsf{b}_2)\} \overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{R}_3} \{1/2{:}\mathsf{f}(\mathcal{O}), 1/2{:}\mathsf{f}(\mathsf{f}(\mathsf{a}))\} \quad (11)$$

to an infinite sequence without normal forms. The resulting chain starts with

$$\begin{array}{ll}
 & \{\quad 1 : \mathsf{c}_1(\mathsf{F}(\mathcal{O}))\} \\
\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{D}_3} & \{\quad 1 : \mathsf{c}_2(\mathsf{F}(\mathsf{a}), \mathsf{A})\} \\
\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{D}_3} & \{\ 1/2 : \mathsf{c}_2(\mathsf{F}(\mathsf{a}), \mathsf{B}_1), 1/2 : \mathsf{c}_2(\mathsf{F}(\mathsf{a}), \mathsf{B}_2)\} \\
\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{R}_3} & \{1/4 : \mathsf{c}_2(\mathsf{F}(\mathsf{b}_1), \mathsf{B}_1), 1/4 : \underline{\mathsf{c}_2(\mathsf{F}(\mathsf{b}_2), \mathsf{B}_1)}, 1/4 : \mathsf{c}_2(\mathsf{F}(\mathsf{b}_1), \mathsf{B}_2), 1/4 : \mathsf{c}_2(\mathsf{F}(\mathsf{b}_2), \mathsf{B}_2).\}
\end{array}$$

The second and third term in the last distribution do not correspond to terms in the original rewrite sequence (11). After the next $\mathcal{D}_3$-step which removes B_1, no further $\mathcal{D}_3$-step can be applied to the underlined term anymore, because b_2 cannot be rewritten to $\mathcal{O}$. Thus, the resulting chain criterion would be unsound, as every chain $(\mu_n)_{n \in \mathbb{N}}$ in this example contains such $\mathcal{D}_3$-normal forms and therefore, it is AST (i.e., $\lim\limits_{n \to \infty} |\mu_n|_{\mathcal{D}_3} = 1$ where $|\mu_n|_{\mathcal{D}_3}$ is the probability for $\mathcal{D}_3$-normal forms in μ_n). So we have to ensure that when A is rewritten to B_1 via a DT from $\mathcal{D}_3$, then the "copy" a of the redex A is rewritten via $\mathcal{R}_3$ to the corresponding term b_1 instead of b_2. Thus, after the step with $\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{R}_3}$ we should have $\mathsf{c}_2(\mathsf{F}(\mathsf{b}_1), \mathsf{B}_1)$ and $\mathsf{c}_2(\mathsf{F}(\mathsf{b}_2), \mathsf{B}_2)$, but not $\mathsf{c}_2(\mathsf{F}(\mathsf{b}_2), \mathsf{B}_1)$ or $\mathsf{c}_2(\mathsf{F}(\mathsf{b}_1), \mathsf{B}_2)$.

Therefore, for our new adaption of DPs to the probabilistic setting, we operate on *pairs*. Instead of having a rule $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\}$ from $\mathcal{R}$ and its corresponding dependency tuple $\ell^\# \to \{p_1 : dp(r_1), \ldots, p_k : dp(r_k)\}$ separately, we couple them together to $\langle \ell^\#, \ell \rangle \to \{p_1 : \langle dp(r_1), r_1 \rangle, \ldots, p_k : \langle dp(r_k), r_k \rangle\}$. This type of rewrite system is called a *probabilistic pair term rewrite system (PPTRS)*, and its rules are called *coupled dependency tuples*. Our new DP framework works on *(probabilistic) DP problems* $(\mathcal{P}, \mathcal{S})$, where $\mathcal{P}$ is a PPTRS and $\mathcal{S}$ is a PTRS.

Definition 22 (Coupled Dependency Tuple). *Let $\mathcal{R}$ be a PTRS. For every $\ell \to \mu = \{p_1 : r_1, \ldots, p_k : r_k\} \in \mathcal{R}$, its* coupled dependency tuple *(or simply* dependency tuple*,* DT*) is $\mathcal{DT}(\ell \to \mu) = \langle \ell^\#, \ell \rangle \to \{p_1 : \langle dp(r_1), r_1 \rangle, \ldots, p_k : \langle dp(r_k), r_k \rangle\}$. The set of all coupled dependency tuples of $\mathcal{R}$ is denoted by $\mathcal{DT}(\mathcal{R})$.*

Example 23. The following PTRS $\mathcal{R}_{\mathsf{pdiv}}$ adapts $\mathcal{R}_{\mathsf{div}}$ to the probabilistic setting.

$$\mathsf{minus}(x, \mathcal{O}) \to \{1 : x\} \quad (12) \qquad \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \to \{1 : \mathsf{minus}(x, y)\} \quad (13)$$

$$\mathsf{div}(\mathcal{O}, \mathsf{s}(y)) \to \{1 : \mathcal{O}\} \quad (14)$$

$$\mathsf{div}(\mathsf{s}(x), \mathsf{s}(y)) \to \{1/2 : \mathsf{div}(\mathsf{s}(x), \mathsf{s}(y)), 1/2 : \mathsf{s}(\mathsf{div}(\mathsf{minus}(x, y), \mathsf{s}(y)))\} \quad (15)$$

In (15), we now do the actual rewrite step with a chance of 1/2 or the terms stay the same. Our new probabilistic DP framework can prove automatically that $\mathcal{R}_{\mathsf{pdiv}}$ is iAST, while (as in the non-probabilistic setting) a direct application of polynomial interpretations via Theorem 17 fails. We get $\mathcal{DT}(\mathcal{R}_{\mathsf{pdiv}}) = \{(16), \ldots, (19)\}$:

$$\langle \mathsf{M}(x, \mathcal{O}), \mathsf{minus}(x, \mathcal{O}) \rangle \to \{1 : \langle \mathsf{c}_0,\, x \rangle\} \quad (16)$$

$$\langle \mathsf{M}(\mathsf{s}(x), \mathsf{s}(y)), \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \rangle \to \{1 : \langle \mathsf{c}_1(\mathsf{M}(x, y)),\, \mathsf{minus}(x, y) \rangle\} \quad (17)$$

$$\langle \mathsf{D}(\mathcal{O}, \mathsf{s}(y)), \mathsf{div}(\mathcal{O}, \mathsf{s}(y)) \rangle \to \{1 : \langle \mathsf{c}_0,\, \mathcal{O} \rangle\} \quad (18)$$

$$\begin{aligned} \langle \mathsf{D}(\mathsf{s}(x), \mathsf{s}(y)), \mathsf{div}(\mathsf{s}(x), \mathsf{s}(y)) \rangle \to \{ & 1/2 : \langle \mathsf{c}_1(\mathsf{D}(\mathsf{s}(x), \mathsf{s}(y))),\, \mathsf{div}(\mathsf{s}(x), \mathsf{s}(y)) \rangle, \\ & 1/2 : \langle \mathsf{c}_2(\mathsf{D}(\mathsf{minus}(x, y), \mathsf{s}(y)), \mathsf{M}(x, y)),\, \mathsf{s}(\mathsf{div}(\mathsf{minus}(x, y), \mathsf{s}(y))) \rangle\} \quad (19) \end{aligned}$$

Definition 24 (PPTRS, $\overset{\mathrm{i}}{\rightarrowtriangle}_{\mathcal{P},\mathcal{S}}$). *Let $\mathcal{P}$ be a finite set of rules of the form $\langle \ell^\#, \ell \rangle \to \{p_1 : \langle d_1, r_1 \rangle, \ldots, p_k : \langle d_k, r_k \rangle\}$. For every such rule, let* $\mathrm{proj}_1(\mathcal{P})$ *contain $\ell^\# \to \{p_1 : d_1, \ldots, p_k : d_k\}$ and let* $\mathrm{proj}_2(\mathcal{P})$ *contain $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\}$. If* $\mathrm{proj}_2(\mathcal{P})$ *is a PTRS and $cont(d_j) \subseteq cont(dp(r_j))$ holds[6] for all $1 \leq j \leq k$, then $\mathcal{P}$ is a* probabilistic pair term rewrite system (PPTRS).

Let $\mathcal{S}$ be a PTRS. Then a normalized term $\mathsf{c}_n(s_1, \ldots, s_n)$ rewrites *with the PPTRS $\mathcal{P}$ to $\{p_1 : b_1, \ldots, p_k : b_k\}$ w.r.t. $\mathcal{S}$ (denoted $\overset{\mathrm{i}}{\rightarrowtriangle}_{\mathcal{P},\mathcal{S}}$) if there are an $1 \leq i \leq n$, an $\langle \ell^\#, \ell \rangle \to \{p_1 : \langle d_1, r_1 \rangle, \ldots, p_k : \langle d_k, r_k \rangle\} \in \mathcal{P}$, a substitution σ with $s_i = \ell^\# \sigma \in \mathtt{NF}_{\mathcal{S}}$, and for all $1 \leq j \leq k$ we have $b_j = \mathsf{c}_n(t_1^j, \ldots, t_n^j)$ where*

[6] The reason for $cont(d_j) \subseteq cont(dp(r_j))$ instead of $cont(d_j) = cont(dp(r_j))$ is that in this way processors can remove terms from the right-hand sides of DTs, see Theorem 32.

- *$t_i^j = d_j\sigma$ for all $1 \leq j \leq k$, i.e., we rewrite the term s_i using* $\mathrm{proj}_1(\mathcal{P})$*.*
- *For every $1 \leq i' \leq n$ with $i \neq i'$ we have*
  (i) *$t_{i'}^j = s_{i'}$ for all $1 \leq j \leq k$ or*
  (ii) *$t_{i'}^j = s_{i'}[r_j\sigma]_\tau$ for all $1 \leq j \leq k$,*
  if $s_{i'}|_\tau = \ell\sigma$ for some position τ and if $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\} \in \mathcal{S}$.
  So $s_{i'}$ stays the same in all b_j or we can apply the rule from $\mathrm{proj}_2(\mathcal{P})$ *to rewrite $s_{i'}$ in all b_j, provided that this rule is also contained in $\mathcal{S}$. Note that even if the rule is applicable, the term $s_{i'}$ can still stay the same in all b_j.*

Example 25. For $\mathcal{R}_3$ from Example 21, the (coupled) dependency tuple for the f-rule is $\langle \mathsf{F}(\mathcal{O}), \mathsf{f}(\mathcal{O})\rangle \to \{1 : \langle \mathsf{c}_2(\mathsf{F}(\mathsf{a}), \mathsf{A}), \mathsf{f}(\mathsf{a})\rangle\}$ and the DT for the a-rule is $\langle \mathsf{A}, \mathsf{a}\rangle \to \{1/2 : \langle \mathsf{c}_1(\mathsf{B}_1), \mathsf{b}_1\rangle, 1/2 : \langle \mathsf{c}_1(\mathsf{B}_2), \mathsf{b}_2\rangle\}$. With the lifting $\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{P},\mathcal{S}}$ of $\overset{\mathsf{i}}{\rightarrow}_{\mathcal{P},\mathcal{S}}$, we get the following sequence which corresponds to the rewrite sequence (11) from Example 21.

$$\begin{aligned} \{1 : \mathsf{c}_1(\mathsf{F}(\mathcal{O}))\} &\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{DT}(\mathcal{R}_3),\mathcal{R}_3} \{1 : \mathsf{c}_2(\mathsf{F}(\mathsf{a}), \mathsf{A})\} \\ &\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{DT}(\mathcal{R}_3),\mathcal{R}_3} \{1/2 : \mathsf{c}_2(\mathsf{F}(\mathsf{b}_1), \mathsf{B}_1), 1/2 : \mathsf{c}_2(\mathsf{F}(\mathsf{b}_2), \mathsf{B}_2)\} \end{aligned} \tag{20}$$

So with the PPTRS, when rewriting A to B_1 in the second step, we can simultaneously rewrite the inner subterm a of $\mathsf{F}(\mathsf{a})$ to b_1 or keep a unchanged, but we cannot rewrite a to b_2. This is ensured by b_1 in the second component of $\langle \mathsf{A}, \mathsf{a}\rangle \to \{1/2 : \langle \mathsf{c}_1(\mathsf{B}_1), \mathsf{b}_1\rangle, \ldots\}$, since by Definition 24, if $s_{i'}$ contains $\ell\sigma$ at some arbitrary position τ, then one can (only) use the rule in the second component of the DT to rewrite $\ell\sigma$ (i.e., here we have $s_{i'} = \mathsf{F}(\mathsf{a})$, $s_i = \mathsf{A}$, and $s_{i'}|_\tau = \mathsf{a}$). A similar observation holds when rewriting A to B_2. Recall that with the notion of chains in Example 21, one *cannot simulate* every possible rewrite sequence, which leads to unsoundness. In contrast, with the notion of coupled DTs and PPTRSs, every possible rewrite sequence *can be simulated* which ensures soundness of the chain criterion. Of course, due to the ambiguity in (i) and (ii) of Definition 24, one could also create other "unsuitable" $\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{DT}(\mathcal{R}_3),\mathcal{R}_3}$-sequences where a is not reduced to b_1 and b_2 in the second step, but is kept unchanged. This does not affect the soundness of the chain criterion, since every rewrite sequence of the original PTRS can be simulated by a "suitable" chain. To obtain completeness of the chain criterion, one would have to avoid such "unsuitable" sequences.

We also introduce an analogous rewrite relation for PTRSs, where we can apply the same rule simultaneously to the same subterms in a single rewrite step.

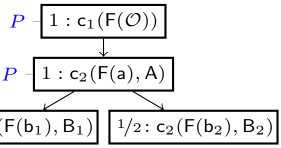
Definition 26 ($\overset{\mathsf{i}}{\rightarrow}_{\mathcal{S}}$). *For a PTRS $\mathcal{S}$ and a normalized term $\mathsf{c}_n(s_1, \ldots, s_n)$, we define $\mathsf{c}_n(s_1, ..., s_n) \overset{\mathsf{i}}{\rightarrow}_{\mathcal{S}} \{p_1 : b_1, ..., p_k : b_k\}$ if there are an $1 \leq i \leq n$, an*

$\ell \to \{p_1 : r_1, \ldots, p_k : r_k\} \in \mathcal{S}$, a position π, a substitution σ with $s_i|_\pi = \ell\sigma$ such that every proper subterm of $\ell\sigma$ is in $\mathtt{NF}_{\mathcal{S}}$, and for all $1 \leq j \leq k$ we have $b_j = \mathsf{c}_n(t_1^j, \ldots, t_n^j)$ where

- *$t_i^j = s_i[r_j\sigma]_\pi$ for all $1 \leq j \leq k$, i.e., we rewrite the term s_i using $\mathcal{S}$.*
- *For every $1 \leq i' \leq n$ with $i \neq i'$ we have*
  (i) *$t_{i'}^j = s_{i'}$ for all $1 \leq j \leq k$ or*
  (ii) *$t_{i'}^j = s_{i'}[r_j\sigma]_\tau$ for all $1 \leq j \leq k$, if $s_{i'}|_\tau = \ell\sigma$ for some position τ.*

So for example, the lifting $\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{S}}$ of $\xrightarrow{\mathsf{i}}_{\mathcal{S}}$ for $\mathcal{S} = \mathcal{R}_3$ rewrites $\{1 : \mathsf{c}_2(\mathsf{f}(\mathsf{a}), \mathsf{a})\}$ to both $\{1/2 : \mathsf{c}_2(\mathsf{f}(\mathsf{b}_1), \mathsf{b}_1), 1/2 : \mathsf{c}_2(\mathsf{f}(\mathsf{b}_2), \mathsf{b}_2)\}$ and $\{1/2 : \mathsf{c}_2(\mathsf{f}(\mathsf{a}), \mathsf{b}_1), 1/2 : \mathsf{c}_2(\mathsf{f}(\mathsf{a}), \mathsf{b}_2)\}$.

A straightforward adaption of "chains" to the probabilistic setting using $\overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{P},\mathcal{S}} \circ \overset{\mathsf{i}}{\rightrightarrows}_{\mathcal{S}}^*$ would force us to use steps with DTs from $\mathcal{P}$ at the same time for all terms in a multi-distribution. Therefore, instead we view a rewrite sequence on multi-distributions as a tree (e.g., the tree representation of the rewrite sequence (20) from Example 25 is on the right). Regarding the paths in this tree (which represent rewrite sequences of terms with certain probabilities) allows us to adapt the idea of chains, i.e., that one uses only finitely many $\mathcal{S}$-steps before the next step with a DT from $\mathcal{P}$.



Definition 27 (Chain Tree). *$\mathfrak{T} = (V, E, L, P)$ is an (innermost) $(\mathcal{P}, \mathcal{S})$*-chain tree *if*

1. *$V \neq \varnothing$ is a possibly infinite set of nodes and $E \subseteq V \times V$ is a set of directed edges, such that (V, E) is a (possibly infinite) directed tree where $vE = \{w \mid (v, w) \in E\}$ is finite for every $v \in V$.*
2. *$L : V \to (0, 1] \times \mathcal{T}\left(\Sigma \uplus \Sigma^{\#}, \mathcal{V}\right)$ labels every node v by a probability p_v and a term t_v. For the root $v \in V$ of the tree, we have $p_v = 1$.*
3. *$P \subseteq V \setminus$ Leaf (where Leaf are all leaves) is a subset of the inner nodes to indicate whether we use the PPTRS $\mathcal{P}$ or the PTRS $\mathcal{S}$ for the rewrite step. $S = V \setminus (\text{Leaf} \cup P)$ are all inner nodes that are not in P. Thus, $V = P \uplus S \uplus$ Leaf.*
4. *For all $v \in P$: If $vE = \{w_1, \ldots, w_k\}$, then $t_v \xrightarrow{\mathsf{i}}_{\mathcal{P},\mathcal{S}} \{\frac{p_{w_1}}{p_v} : t_{w_1}, \ldots, \frac{p_{w_k}}{p_v} : t_{w_k}\}$.*
5. *For all $v \in S$: If $vE = \{w_1, \ldots, w_k\}$, then $t_v \xrightarrow{\mathsf{i}}_{\mathcal{S}} \{\frac{p_{w_1}}{p_v} : t_{w_1}, \ldots, \frac{p_{w_k}}{p_v} : t_{w_k}\}$.*
6. *Every infinite path in $\mathfrak{T}$ contains infinitely many nodes from P.*

Conditions 1–5 ensure that the tree represents a valid rewrite sequence and the last condition is the main property for chains.

Definition 28 ($|\mathfrak{T}|_{\text{Leaf}}$, iAST). *For any innermost $(\mathcal{P}, \mathcal{S})$-chain tree $\mathfrak{T}$ we define $|\mathfrak{T}|_{\text{Leaf}} = \sum_{v \in \text{Leaf}} p_v$. We say that $(\mathcal{P}, \mathcal{S})$ is* iAST *if we have $|\mathfrak{T}|_{\text{Leaf}} = 1$ for every innermost $(\mathcal{P}, \mathcal{S})$-chain tree $\mathfrak{T}$.*

While we have $|\mathfrak{T}|_{\text{Leaf}} = 1$ for every finite chain tree $\mathfrak{T}$, for infinite chain trees $\mathfrak{T}$ we may have $|\mathfrak{T}|_{\text{Leaf}} < 1$ or even $|\mathfrak{T}|_{\text{Leaf}} = 0$ if $\mathfrak{T}$ has no leaf at all.

With this new type of DTs and chain trees, we now obtain an analogous chain criterion to the non-probabilistic setting.

Theorem 29 (Chain Criterion). *A PTRS $\mathcal{R}$ is iAST if $(\mathcal{DT}(\mathcal{R}), \mathcal{R})$ is iAST.*

In contrast to the non-probabilistic case, our chain criterion as presented in the paper is *sound* but not *complete* (i.e., we do not have "iff" in Theorem 29). However, we also developed a refinement where our chain criterion is made complete by also storing the positions of the defined symbols in $dp(r)$ [27]. In this way, one can avoid "unsuitable" chain trees, as discussed at the end of Example 25.

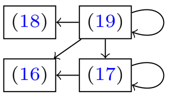
Our notion of DTs and chain trees is only suitable for *innermost* evaluation. To see this, consider the PTRSs $\mathcal{R}_1'$ and $\mathcal{R}_2'$ which both contain $\mathsf{g} \to \{1/2 : \mathcal{O}, 1/2 : \mathsf{h}(\mathsf{g})\}$, but in addition $\mathcal{R}_1'$ has the rule $\mathsf{h}(x) \to \{1 : \mathsf{f}(x,x)\}$ and $\mathcal{R}_2'$ has the rule $\mathsf{h}(x) \to \{1 : \mathsf{f}(x,x,x)\}$. Similar to $\mathcal{R}_1$ and $\mathcal{R}_2$ in (10), $\mathcal{R}_1'$ is AST while $\mathcal{R}_2'$ is not. In contrast, both $\mathcal{R}_1'$ and $\mathcal{R}_2'$ are iAST, since the innermost evaluation strategy prevents the application of the h-rule to terms containing g. Our DP framework handles $\mathcal{R}_1'$ and $\mathcal{R}_2'$ in the same way, as both have the same DT $\langle \mathsf{G}, \mathsf{g} \rangle \to \{1/2 : \langle \mathsf{c}_0, \mathcal{O} \rangle, 1/2 : \langle \mathsf{c}_2(\mathsf{H}(\mathsf{g}), \mathsf{G}), \mathsf{h}(\mathsf{g}) \rangle\}$ and a DT $\langle \mathsf{H}(x), \mathsf{h}(x) \rangle \to \{1 : \langle \mathsf{c}_0, \mathsf{f}(\ldots) \rangle\}$. Even if we allowed the application of the second DT to terms of the form $\mathsf{H}(\mathsf{g})$, we would still obtain $|\mathfrak{T}|_{\text{Leaf}} = 1$ for every chain tree $\mathfrak{T}$. So a DP framework to analyze "full" instead of innermost AST would be considerably more involved.

4.2 The Probabilistic DP Framework

Now we introduce the probabilistic dependency pair framework which keeps the core ideas of the non-probabilistic framework. So instead of applying one ordering for a PTRS directly as in Theorem 17, we want to benefit from modularity. Now a *DP processor* Proc is of the form $\text{Proc}(\mathcal{P}, \mathcal{S}) = \{(\mathcal{P}_1, \mathcal{S}_1), \ldots, (\mathcal{P}_n, \mathcal{S}_n)\}$, where $\mathcal{P}, \mathcal{P}_1, \ldots, \mathcal{P}_n$ are PPTRSs and $\mathcal{S}, \mathcal{S}_1, \ldots, \mathcal{S}_n$ are PTRSs. A processor Proc is *sound* if $(\mathcal{P}, \mathcal{S})$ is iAST whenever $(\mathcal{P}_i, \mathcal{S}_i)$ is iAST for all $1 \leq i \leq n$. It is *complete* if $(\mathcal{P}_i, \mathcal{S}_i)$ is iAST for all $1 \leq i \leq n$ whenever $(\mathcal{P}, \mathcal{S})$ is iAST. In the following, we adapt the three main processors from Theorems 5, 6, and 7 to the probabilistic setting and present two additional processors.

The (innermost) $(\mathcal{P}, \mathcal{S})$-*dependency graph* indicates which DTs from $\mathcal{P}$ can rewrite to each other using the PTRS $\mathcal{S}$. The possibility of rewriting with $\mathcal{S}$ is not related to the probabilities. Thus, for the dependency graph, we can use the *non-probabilistic variant* $\text{np}(\mathcal{S}) = \{\ell \to r_j \mid \ell \to \{p_1 : r_1, \ldots, p_k : r_k\} \in \mathcal{S}, 1 \leq j \leq k\}$.

Definition 30 (Dep. Graph). *The node set of the* $(\mathcal{P}, \mathcal{S})$-dependency graph *is $\mathcal{P}$ and there is an edge from $\langle \ell_1^{\#}, \ell_1 \rangle \to \{p_1 : \langle d_1, r_1 \rangle, \ldots, p_k : \langle d_k, r_k \rangle\}$ to $\langle \ell_2^{\#}, \ell_2 \rangle \to \ldots$ if there are substitutions σ_1, σ_2 and $t^{\#} \in cont(d_j)$ for some $1 \leq j \leq k$ such that $t^{\#}\sigma_1 \xrightarrow{\mathsf{i}}{}^{*}_{\text{np}(\mathcal{S})} \ell_2^{\#}\sigma_2$ and both $\ell_1^{\#}\sigma_1$ and $\ell_2^{\#}\sigma_2$ are in $\mathtt{NF}_{\mathcal{S}}$.*



For $\mathcal{R}_{\mathsf{pdiv}}$ from Example 23, the $(\mathcal{DT}(\mathcal{R}_{\mathsf{pdiv}}), \mathcal{R}_{\mathsf{pdiv}})$-dependency graph is on the side. In the non-probabilistic DP framework, every step with $\xrightarrow{\mathrm{i}}_{\mathcal{D},\mathcal{R}}$ corresponds to an edge in the $(\mathcal{D}, \mathcal{R})$-dependency graph. Similarly, in the probabilistic setting, every path from one node of P to the next node of P in a $(\mathcal{P}, \mathcal{S})$-chain tree corresponds to an edge in the $(\mathcal{P}, \mathcal{S})$- dependency graph. Since every infinite path in a chain tree contains infinitely many nodes from P, when tracking the arguments of the compound symbols, every such path traverses a cycle of the dependency graph infinitely often. Thus, it again suffices to consider the SCCs of the dependency graph separately. So for our example, we obtain $\mathrm{Proc}_{\mathtt{DG}}(\mathcal{DT}(\mathcal{R}_{\mathsf{pdiv}}), \mathcal{R}_{\mathsf{pdiv}}) = \{(\{(17)\}, \mathcal{R}_{\mathsf{pdiv}}), (\{(19)\}, \mathcal{R}_{\mathsf{pdiv}})\}$. To automate the following two processors, the same over-approximation techniques as for the non-probabilistic dependency graph can be used.

Theorem 31 (Prob. Dep. Graph Processor). *For the SCCs $\mathcal{P}_1, ..., \mathcal{P}_n$ of the $(\mathcal{P}, \mathcal{S})$-dependency graph,* $\mathrm{Proc}_{\mathtt{DG}}(\mathcal{P}, \mathcal{S}) = \{(\mathcal{P}_1, \mathcal{S}), ..., (\mathcal{P}_n, \mathcal{S})\}$ *is sound and complete.*

Next, we introduce a new *usable terms processor* (a similar processor was also proposed for the DTs in [37]). Since we regard dependency *tuples* instead of pairs, after applying $\mathrm{Proc}_{\mathtt{DG}}$, the right-hand sides of DTs $\langle \ell_1^{\#}, \ell_1 \rangle \to \ldots$ might still contain terms $t^{\#}$ where no instance $t^{\#}\sigma_1$ rewrites to an instance $\ell_2^{\#}\sigma_2$ of a left-hand side of a DT (where we only consider instantiations such that $\ell_1^{\#}\sigma_1$ and $\ell_2^{\#}\sigma_2$ are in $\mathtt{NF}_{\mathcal{S}}$, because only such instantiations are regarded in chain trees). Then $t^{\#}$ can be removed from the right-hand side of the DT. For example, in the DP problem $(\{(19)\}, \mathcal{R}_{\mathsf{pdiv}})$, the only DT (19) has the left-hand side $\mathsf{D}(\mathsf{s}(x), \mathsf{s}(y))$. As the term $\mathsf{M}(x, y)$ in (19)'s right-hand side cannot "reach" $\mathsf{D}(\ldots)$, the following processor removes it, i.e., $\mathrm{Proc}_{\mathtt{UT}}(\{(19)\}, \mathcal{R}_{\mathsf{pdiv}}) = \{(\{(21)\}, \mathcal{R}_{\mathsf{pdiv}})\}$, where (21) is

$$\begin{aligned} \langle \mathsf{D}(\mathsf{s}(x), \mathsf{s}(y)), \mathsf{div}(\mathsf{s}(x), \mathsf{s}(y)) \rangle \to \{ {}^{1}\!/_{2} : \langle \mathsf{c}_1(\mathsf{D}(\mathsf{s}(x), \mathsf{s}(y))), \mathsf{div}(\mathsf{s}(x), \mathsf{s}(y)) \rangle, \\ {}^{1}\!/_{2} : \langle \mathsf{c}_1(\mathsf{D}(\mathsf{minus}(x, y), \mathsf{s}(y))), \mathsf{s}(\mathsf{div}(\mathsf{minus}(x, y), \mathsf{s}(y))) \rangle \}. \end{aligned} \quad (21)$$

So both Theorems 31 and 32 are needed to fully simulate the dependency graph processor in the probabilistic setting, i.e., they are both necessary to guarantee that the probabilistic DP processors work analogously to the non-probabilistic ones (which in turn ensures that the probabilistic DP framework is similar in power to its non-probabilistic counterpart). This is also confirmed by our experiments in Sect. 5 which show that disabling the processor of Theorem 32 affects the power of our approach. For example, without Theorem 32, the proof that $\mathcal{R}_{\mathsf{pdiv}}$ is iAST in the probabilistic DP framework would require a more complicated polynomial interpretation. In contrast, when using both processors of Theorems 31 and 32, then one can prove iAST of $\mathcal{R}_{\mathsf{pdiv}}$ with the same polynomial interpretation that was used to prove iTerm of $\mathcal{R}_{\mathsf{div}}$ (see Example 36).

Theorem 32 (Usable Terms Processor). *Let $\ell_1^\#$ be a term and $(\mathcal{P},\mathcal{S})$ be a DP problem. We call a term $t^\#$* usable *w.r.t. $\ell_1^\#$ and $(\mathcal{P},\mathcal{S})$ if there is a $\langle \ell_2^\#, \ell_2\rangle \to \ldots \in \mathcal{P}$ and substitutions σ_1, σ_2 such that $t^\# \sigma_1 \xrightarrow{\mathsf{i}}{}^*_{\mathrm{np}(\mathcal{S})} \ell_2^\# \sigma_2$ and both $\ell_1^\# \sigma_1$ and $\ell_2^\# \sigma_2$ are in $\mathtt{NF}_{\mathcal{S}}$. If $d = \mathsf{c}_n(t_1^\#, \ldots, t_n^\#)$, then $\mathcal{UT}(d)_{\ell_1^\#,\mathcal{P},\mathcal{S}}$ denotes the term $\mathsf{c}_m(t_{i_1}^\#, \ldots, t_{i_m}^\#)$, where $1 \leq i_1 < \ldots < i_m \leq n$ are the indices of all terms $t_i^\#$ that are usable w.r.t. $\ell_1^\#$ and $(\mathcal{P},\mathcal{S})$. The transformation that removes all non-usable terms in the right-hand sides of dependency tuples is denoted by:*

$$\begin{aligned}\mathcal{T}_{\mathtt{UT}}(\mathcal{P},\mathcal{S}) = \{\langle \ell^\#, \ell\rangle \to \{p_1 : \langle \mathcal{UT}(d_1)_{\ell^\#,\mathcal{P},\mathcal{S}}, r_1\rangle, \ldots, p_k : \langle \mathcal{UT}(d_k)_{\ell^\#,\mathcal{P},\mathcal{S}}, r_k\rangle\} \\ \mid \langle \ell^\#, \ell\rangle \to \{p_1 : \langle d_1, r_1\rangle, \ldots, p_k : \langle d_k, r_k\rangle\} \in \mathcal{P}\}\end{aligned}$$

Then $\mathrm{Proc}_{\mathtt{UT}}(\mathcal{P},\mathcal{S}) = \{(\mathcal{T}_{\mathtt{UT}}(\mathcal{P},\mathcal{S}),\mathcal{S})\}$ *is sound and complete.*

To adapt the *usable rules processor*, we adjust the definition of usable rules such that it regards every term in the support of the distribution on the right-hand side of a rule. The usable rules processor only deletes non-usable rules from $\mathcal{S}$, but not from $\mathrm{proj}_2(\mathcal{P})$. This is sufficient, because according to Definition 24, rules from $\mathrm{proj}_2(\mathcal{P})$ can only be applied if they also occur in $\mathcal{S}$.

Theorem 33 (Probabilistic Usable Rules Processor). *Let $(\mathcal{P},\mathcal{S})$ be a DP problem. For every $f \in \Sigma \uplus \Sigma^\#$ let* $\mathrm{Rules}_{\mathcal{S}}(f) = \{\ell \to \mu \in \mathcal{S} \mid \mathrm{root}(\ell) = f\}$. *For any term $t \in \mathcal{T}\left(\Sigma \uplus \Sigma^\#, \mathcal{V}\right)$, its* usable rules *$\mathcal{U}_{\mathcal{S}}(t)$ are the smallest set such that $\mathcal{U}_{\mathcal{S}}(x) = \varnothing$ for all $x \in \mathcal{V}$ and* $\mathcal{U}_{\mathcal{S}}(f(t_1, \ldots, t_n)) = \mathrm{Rules}_{\mathcal{S}}(f) \cup \bigcup_{i=1}^n \mathcal{U}_{\mathcal{S}}(t_i) \cup \bigcup_{\ell \to \mu \in \mathrm{Rules}_{\mathcal{S}}(f), r \in \mathrm{Supp}(\mu)} \mathcal{U}_{\mathcal{S}}(r)$. *The* usable rules *for $(\mathcal{P},\mathcal{S})$ are* $\mathcal{U}(\mathcal{P},\mathcal{S}) = \bigcup_{\ell^\# \to \mu \in \mathrm{proj}_1(\mathcal{P}), d \in \mathrm{Supp}(\mu)} \mathcal{U}_{\mathcal{S}}(d)$. *Then* $\mathrm{Proc}_{\mathtt{UR}}(\mathcal{P},\mathcal{S}) = \{(\mathcal{P},\mathcal{U}(\mathcal{P},\mathcal{S}))\}$ *is sound.*

Example 34. For the DP problem $(\{(21)\}, \mathcal{R}_{\mathsf{pdiv}})$ only the minus-rules are usable and thus $\mathrm{Proc}_{\mathtt{UR}}(\{(21)\}, \mathcal{R}_{\mathsf{pdiv}}) = \{(\{(21)\}, \{(12),(13)\})\}$. For $(\{(17)\}, \mathcal{R}_{\mathsf{pdiv}})$ there are no usable rules at all, hence $\mathrm{Proc}_{\mathtt{UR}}(\{(17)\}, \mathcal{R}_{\mathsf{pdiv}}) = \{(\{(17)\}, \varnothing)\}$.

For the *reduction pair processor*, we again restrict ourselves to multilinear polynomials and use analogous constraints as in our new criterion for the direct application of polynomial interpretations to PTRSs (Theorem 17), but adapted to DP problems $(\mathcal{P},\mathcal{S})$. Moreover, as in the original reduction pair processor of Theorem 7, the polynomials only have to be weakly monotonic. For every rule in $\mathcal{S}$ or $\mathrm{proj}_1(\mathcal{P})$, we require that the expected value is weakly decreasing. The reduction pair processor then removes those DTs $\langle \ell^\#, \ell\rangle \to \{p_1 : \langle d_1, r_1\rangle, \ldots, p_k : \langle d_k, r_k\rangle\}$ from $\mathcal{P}$ where in addition there is at least one term d_j that is strictly decreasing. Recall that we can also rewrite with the original rule $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\}$ from $\mathrm{proj}_2(\mathcal{P})$, provided that it is also contained in $\mathcal{S}$. Therefore, to remove the dependency tuple, we also have to require that the rule $\ell \to r_j$ is weakly decreasing. Finally, we have to use c-*additive* interpretations (with $\mathsf{c}_{n\,\mathrm{Pol}}(x_1, \ldots, x_n) = x_1 + \ldots + x_n$) to handle compound symbols and their normalization correctly.

Theorem 35 (Probabilistic Reduction Pair Processor). *Let* $\mathrm{Pol} : \mathcal{T}(\Sigma \uplus \Sigma^{\#}, \mathcal{V}) \to \mathbb{N}[\mathcal{V}]$ *be a weakly monotonic, multilinear, and* c*-additive polynomial interpretation. Let* $\mathcal{P} = \mathcal{P}_{\geq} \uplus \mathcal{P}_{>}$ *with* $\mathcal{P}_{>} \neq \varnothing$ *such that:*

(1) For every $\ell \to \{p_1 : r_1, ..., p_k : r_k\} \in \mathcal{S}$*, we have* $\mathrm{Pol}(\ell) \geq \sum_{1 \leq j \leq k} p_j \cdot \mathrm{Pol}(r_j)$*.*
(2) For every $\langle \ell^{\#}, \ell \rangle \to \{p_1 : \langle d_1, r_1 \rangle, \ldots, p_k : \langle d_k, r_k \rangle\} \in \mathcal{P}$*, we have* $\mathrm{Pol}(\ell^{\#}) \geq \sum_{1 \leq j \leq k} p_j \cdot \mathrm{Pol}(d_j)$*.*
(3) For every $\langle \ell^{\#}, \ell \rangle \to \{p_1 : \langle d_1, r_1 \rangle, \ldots, p_k : \langle d_k, r_k \rangle\} \in \mathcal{P}_{>}$*, there exists a* $1 \leq j \leq k$ *with* $\mathrm{Pol}(\ell^{\#}) > \mathrm{Pol}(d_j)$*.*
If $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\} \in \mathcal{S}$*, then we additionally have* $\mathrm{Pol}(\ell) \geq \mathrm{Pol}(r_j)$*.*

Then $\mathrm{Proc}_{\mathtt{RP}}(\mathcal{P}, \mathcal{S}) = \{(\mathcal{P}_{\geq}, \mathcal{S})\}$ *is sound and complete.*

Example 36. The constraints of the reduction pair processor for the two DP problems from Example 34 are satisfied by the c-additive polynomial interpretation which again maps $\mathcal{O}$ to 0, $\mathsf{s}(x)$ to $x + 1$, and all other non-constant function symbols to the projection on their first arguments. As in the non-probabilistic case, this results in DP problems of the form $(\varnothing, \ldots)$ and subsequently, $\mathrm{Proc}_{\mathtt{DG}}(\varnothing, \ldots)$ yields $\varnothing$. By the soundness of all processors, this proves that $\mathcal{R}_{\mathsf{pdiv}}$ is iAST.

So with the new probabilistic DP framework, the proof that $\mathcal{R}_{\mathsf{pdiv}}$ is iAST is analogous to the proof that $\mathcal{R}_{\mathsf{div}}$ is iTerm in the original DP framework (the proofs even use the same polynomial interpretation in the respective reduction pair processors). This indicates that our novel framework for PTRSs has the same essential concepts and advantages as the original DP framework for TRSs. This is different from our previous adaption of dependency pairs for complexity analysis of TRSs, which also relies on dependency tuples [37]. There, the power is considerably restricted, because one does not have full modularity as one cannot decompose the proof according to the SCCs of the dependency graph.

In proofs with the probabilistic DP framework, one may obtain DP problems $(\mathcal{P}, \mathcal{S})$ that have a non-probabilistic structure (i.e., every DT in $\mathcal{P}$ has the form $\langle \ell^{\#}, \ell \rangle \to \{1 : \langle d, r \rangle\}$ and every rule in $\mathcal{S}$ has the form $\ell' \to \{1 : r'\}$). We now introduce a processor that allows us to switch to the original non-probabilistic DP framework for such (sub-)problems. This is advantageous, because due to the use of dependency *tuples* instead of pairs in $\mathcal{P}$, in general the constraints of the probabilistic reduction pair processor of Theorem 35 are harder than the ones of the reduction pair processor of Theorem 7. Moreover, Theorem 7 is not restricted to multilinear polynomial interpretations and the original DP framework has many additional processors that have not yet been adapted to the probabilistic setting.

Theorem 37. (Probability Removal Processor). *Let* $(\mathcal{P}, \mathcal{S})$ *be a probabilistic DP problem where every DT in* $\mathcal{P}$ *has the form* $\langle \ell^{\#}, \ell \rangle \to \{1 : \langle d, r \rangle\}$ *and every rule in* $\mathcal{S}$ *has the form* $\ell' \to \{1 : r'\}$*. Let* $\mathrm{np}(\mathcal{P}) = \{\ell^{\#} \to t^{\#} \mid \ell^{\#} \to \{1 : d\} \in \mathrm{proj}_1(\mathcal{P}), t^{\#} \in \mathit{cont}(d)\}$*. Then* $(\mathcal{P}, \mathcal{S})$ *is iAST iff the non-probabilistic DP problem* $(\mathrm{np}(\mathcal{P}), \mathrm{np}(\mathcal{S}))$ *is iTerm. So if* $(\mathrm{np}(\mathcal{P}), \mathrm{np}(\mathcal{S}))$ *is iTerm, then the processor* $\mathrm{Proc}_{\mathtt{PR}}(\mathcal{P}, \mathcal{S}) = \varnothing$ *is sound and complete.*

5 Conclusion and Evaluation

Starting with a new "direct" technique to prove almost-sure termination of probabilistic TRSs (Theorem 17), we presented the first adaption of the dependency pair framework to the probabilistic setting in order to prove innermost AST automatically. This is not at all obvious, since most straightforward ideas for such an adaption are unsound (as discussed in Sect. 4.1). So the challenge was to find a suitable definition of dependency pairs (resp. tuples) and chains (resp. chain trees) such that one can define DP processors which are sound and work analogously to the non-probabilistic setting (in order to obtain a framework which is similar in power to the non-probabilistic one). While the soundness proofs for our new processors are much more involved than in the non-probabilistic case, the new processors themselves are quite analogous to their non-probabilistic counterparts and thus, adapting an existing implementation of the non-probabilistic DP framework to the probabilistic one does not require much effort.

We implemented our contributions in our termination prover AProVE, which yields the first tool to prove almost-sure innermost termination of PTRSs on arbitrary data structures (including PTRSs that are not PAST). In our experiments, we compared the direct application of polynomials for proving AST (via our new Theorem 17) with the probabilistic DP framework. We evaluated AProVE on a collection of 67 PTRSs which includes many typical probabilistic algorithms. For example, it contains the following PTRS $\mathcal{R}_{\mathsf{qs}}$ for probabilistic quicksort.

$$\begin{aligned}
\mathsf{rotate}(\mathsf{cons}(x, xs)) &\to \{{}^1\!/\!{}_2 : \mathsf{cons}(x, xs), {}^1\!/\!{}_2 : \mathsf{rotate}(\mathsf{app}(xs, \mathsf{cons}(x, \mathsf{nil})))\}\\
\mathsf{qs}(\mathsf{nil}) &\to \{1 : \mathsf{nil}\}\\
\mathsf{qs}(\mathsf{cons}(x, xs)) &\to \{1 : \mathsf{qsHelp}(\mathsf{rotate}(\mathsf{cons}(x, xs)))\}\\
\mathsf{qsHelp}(\mathsf{cons}(x, xs)) &\to \{1 : \mathsf{app}(\mathsf{qs}(\mathsf{low}(x, xs)), \mathsf{cons}(x, \mathsf{qs}(\mathsf{high}(x, xs))))\}
\end{aligned}$$

The rotate-rules rotate a list randomly often (they are AST, but not terminating). Thus, by choosing the first element of the resulting list, one obtains a random pivot element for the recursive call of quicksort. In addition to the rules above, $\mathcal{R}_{\mathsf{qs}}$ contains rules for list concatenation (app), and rules such that $\mathsf{low}(x, xs)$ (resp. $\mathsf{high}(x, xs)$) returns all elements of the list xs that are smaller (resp. greater or equal) than x, see [28]. Using the probabilistic DP framework, AProVE can prove iAST of $\mathcal{R}_{\mathsf{qs}}$ and many other typical programs.

61 of the 67 examples in our collection are iAST and AProVE can prove iAST for 53 (87%) of them. Here, the DP framework proves iAST for 51 examples and the direct application of polynomial interpretations via Theorem 17 succeeds for 27 examples. (In contrast, proving PAST via the direct application of polynomial interpretations as in [3] only works for 22 examples.) The average runtime of AProVE per example was 2.88 s (where no example took longer than 8 s). So our experiments indicate that the power of the DP framework can now also be used for probabilistic TRSs.

We also performed experiments where we disabled individual processors of the probabilistic DP framework. More precisely, we disabled either the usable terms processor (Theorem 32), both the dependency graph and the usable terms processor (Theorems 31 and 32), or all processors except the reduction pair processor of Theorem 35. Our experiments show that disabling processors indeed affects the power of the approach, in particular for larger examples with several defined symbols (e.g., then AProVE cannot prove iAST of $\mathcal{R}_{\mathsf{qs}}$ anymore). So all of our processors are needed to obtain a powerful technique for termination analysis of PTRSs.

Due to the use of dependency *tuples* instead of pairs, the probabilistic DP framework does not (yet) subsume the direct application of polynomials completely (two examples in our collection can only be proved by the latter, see [28]). Therefore, currently AProVE uses the direct approach of Theorem 17 in addition to the probabilistic DP framework. In future work, we will adapt further processors of the original DP framework to the probabilistic setting, which will also allow us to integrate the direct approach of Theorem 17 into the probabilistic DP framework in a modular way. Moreover, we will develop processors to prove AST of full (instead of innermost) rewriting. Further work may also include processors to disprove (i)AST and possible extensions to analyze PAST and expected runtimes as well. Finally, one could also modify the formalism of PTRSs in order to allow non-constant probabilities which depend on the sizes of terms.

For details on our experiments and for instructions on how to run our implementation in AProVE via its *web interface* or locally, we refer to https://aprove-developers.github.io/ProbabilisticTermRewriting/.

Acknowledgements. We are grateful to Marcel Hark, Dominik Meier, and Florian Frohn for help and advice.

References

1. Agrawal, S., Chatterjee, K., Novotný, P.: Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. Proc. ACM Program. Lang. **2**(POPL), 1–32 (2017). https://doi.org/10.1145/3158122
2. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theor. Comput. Sci. **236**(1–2), 133–178 (2000). https://doi.org/10.1016/S0304-3975(99)00207-8
3. Avanzini, M., Dal Lago, U., Yamada, A.: On probabilistic term rewriting. Sci. Comput. Program. **185** (2020). https://doi.org/10.1016/j.scico.2019.102338
4. Avanzini, M., Moser, G., Schaper, M.: A modular cost analysis for probabilistic programs. Proc. ACM Program. Lang. **4**(OOPSLA), 1–30 (2020). https://doi.org/10.1145/3428240
5. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998). https://doi.org/10.1017/CBO9781139172752
6. Batz, K., Kaminski, B.L., Katoen, J.-P., Matheja, C., Verscht, L.: A calculus for amortized expected runtimes. Proc. ACM Program. Lang. **7**(POPL), 1957–1986 (2023). https://doi.org/10.1145/3571260

7. Beutner, R., Ong, L.: On probabilistic termination of functional programs with continuous distributions. In: Freund, S.N., Yahav, E. (eds.) PLDI 2021, pp. 1312–1326 (2021). https://doi.org/10.1145/3453483.3454111
8. Bournez, O., Kirchner, C.: Probabilistic rewrite strategies. Applications to ELAN. In: Tison, S. (ed.) RTA 2002. LNCS, vol. 2378, pp. 252–266. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45610-4_18
9. Bournez, O., Garnier, F.: Proving positive almost-sure termination. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 323–337. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_24
10. Brockschmidt, M., Musiol, R., Otto, C., Giesl, J.: Automated termination proofs for Java programs with cyclic data. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 105–122. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_13
11. Chatterjee, K., Fu, H., Novotný, P.: Termination analysis of probabilistic programs with martingales. In: Barthe, G., Katoen, J.-P., Silva, A. (eds.) Foundations of Probabilistic Programming, pp. 221–258. Cambridge University Press (2020). https://doi.org/10.1017/9781108770750.008
12. Dal Lago, U., Grellois, C.: Probabilistic termination by monadic affine sized typing. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 393–419. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_15
13. Dershowitz, N.: Orderings for term-rewriting systems. Theor. Comput. Sci. **17**, 279–301 (1982). https://doi.org/10.1016/0304-3975(82)90026-3
14. Faggian, C.: Probabilistic rewriting and asymptotic behaviour: on termination and unique normal forms. Log. Methods Comput. Sci. **18**(2) (2022). https://doi.org/10.46298/lmcs-18(2:5)2022
15. Ferrer Fioriti, L.M., Hermanns, H.: Probabilistic termination: soundness, completeness, and compositionality. In: Rajamani, S.K., Walker, D. (eds.) POPL 2015, pp. 489–501 (2015). https://doi.org/10.1145/2676726.2677001
16. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework: combining techniques for automated termination proofs. In: Baader, F., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3452, pp. 301–331. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32275-7_21
17. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. J. Autom. Reason. **37**(3), 155–203 (2006). https://doi.org/10.1007/s10817-006-9057-7
18. Giesl, J., Raffelsieper, M., Schneider-Kamp, P., Swiderski, S., Thiemann, R.: Automated termination proofs for Haskell by term rewriting. ACM Trans. Program. Lang. Syst. **33**(2), 1–39 (2011). https://doi.org/10.1145/1890028.1890030
19. Giesl, J., Ströder, T., Schneider-Kamp, P., Emmes, F., Fuhs, C.: Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In: De Schreye, D., Janssens, G., King, A. (eds.) PPDP 2012, pp. 1–12 (2012). https://doi.org/10.1145/2370776.2370778
20. Giesl, J., et al.: Analyzing program termination and complexity automatically with AProVE. J. Autom. Reason. **58**(1), 3–31 (2017). https://doi.org/10.1007/s10817-016-9388-y
21. Giesl, J., Giesl, P., Hark, M.: Computing expected runtimes for constant probability programs. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 269–286. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_16
22. Gutiérrez, R., Lucas, S.: MU-TERM: verify termination properties automatically (system description). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020.

LNCS (LNAI), vol. 12167, pp. 436–447. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_28

23. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. Inf. Comput. **199**(1–2), 172–199 (2005). https://doi.org/10.1016/j.ic.2004.10.004
24. Huang, M., Fu, H., Chatterjee, K., Goharshady, A.K.: Modular verification for almost-sure termination of probabilistic programs. Proc. ACM Program. Lang. **3**(OOPSLA), 1–29 (2019). https://doi.org/10.1145/3360555
25. Kaminski, B.L., Katoen, J.-P., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected runtimes of randomized algorithms. J. ACM **65**, 1–68 (2018). https://doi.org/10.1145/3208102
26. Kaminski, B.L., Katoen, J.-P., Matheja, C.: Expected runtime analyis by program verification. In: Barthe, G., Katoen, J.-P., Silva, A. (eds.) Foundations of Probabilistic Programming, pp. 185–220. Cambridge University Press (2020). https://doi.org/10.1017/9781108770750.007
27. Kassing, J.-C.: Using dependency pairs for proving almost-sure termination of probabilistic term rewriting. MA thesis. RWTH Aachen University (2022). https://verify.rwth-aachen.de/da/Kassing-Masterthesis.pdf
28. Kassing, J.-C., Giesl, J.: Proving almost-sure innermost termination of probabilistic term rewriting using dependency pairs. CoRR abs/2305.11741 (2023). https://doi.org/10.48550/arXiv.2305.11741
29. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean termination tool 2. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 295–304. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02348-4_21
30. Lankford, D.S.: On proving term rewriting systems are noetherian. Memo MTP-3, Mathematics Department, Louisiana Technical University, Ruston, LA (1979). http://www.ens-lyon.fr/LIP/REWRITING/TERMINATION/LankfordPolyTerm.pdf
31. Leutgeb, L., Moser, G., Zuleger, F.: Automated expected amortised cost analysis of probabilistic data structures. In: Shoham, S., Vizel, Y. (eds.) CAV 2022. LNCS, vol. 13372, pp. 70–91. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13188-2_4
32. McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.-P.: A new proof rule for almost-sure termination. Proc. ACM Program. Lang. **2**(POPL), 1–28 (2018). https://doi.org/10.1145/3158121
33. Meyer, F., Hark, M., Giesl, J.: Inferring expected runtimes of probabilistic integer programs using expected sizes. In: TACAS 2021. LNCS, vol. 12651, pp. 250–269. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_14
34. Moosbrugger, M., Bartocci, E., Katoen, J.-P., Kovács, L.: Automated termination analysis of polynomial probabilistic programs. In: ESOP 2021. LNCS, vol. 12648, pp. 491–518. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72019-3_18
35. Moser, G., Schaper, M.: From `Jinja` bytecode to term rewriting: a complexity reflecting transformation. Inf. Comput. **261**, 116–143 (2018). https://doi.org/10.1016/j.ic.2018.05.007
36. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: resource analysis for probabilistic programs. In: Foster, J.S., Grossman, D. (eds.) PLDI 2018, pp. 496–512 (2018). https://doi.org/10.1145/3192366.3192394
37. Noschinski, L., Emmes, F., Giesl, J.: Analyzing innermost runtime complexity of term rewriting by dependency pairs. J. Autom. Reason. **51**, 27–56 (2013). https://doi.org/10.1007/978-3-642-22438-6_32

38. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Automated termination analysis of Java bytecode by term rewriting. In: Lynch, C. (ed.) RTA 2010. LIPIcs 6, pp. 259–276 (2010). https://doi.org/10.4230/LIPIcs.RTA.2010.259
39. Wang, D., Kahn, D.M., Hoffmann, J.: Raising expectations: automating expected cost analysis with types. Proc. ACM Program. Lang. **4**(ICFP), 1–31 (2020). https://doi.org/10.1145/3408992
40. Yamada, A., Kusakari, K., Sakabe, T.: Nagoya termination tool. In: Dowek, G. (ed.) RTA-TLCA 2014. LNCS, vol. 8560, pp. 466–475. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_32

Verification of NP-Hardness Reduction Functions for Exact Lattice Problems

Katharina Kreuzer(✉) and Tobias Nipkow

Technical University of Munich, Boltzmannstr. 3, 85748 Garching, Germany
k.kreuzer@tum.de

Abstract. This paper describes the formal verification of NP-hardness reduction functions of two key problems relevant in algebraic lattice theory: the closest vector problem and the shortest vector problem, both in the infinity norm. The formalization uncovered a number of problems with the existing proofs in the literature. The paper describes how these problems were corrected in the formalization. The work was carried out in the proof assistant Isabelle.

Keywords: verification · NP-hardness · lattice problems · integer programming

1 Introduction

In recent years, algebraic lattices have received increasing attention for their use in post-quantum cryptography. Algebraic lattices are additive, discrete subgroups of $\mathbb{R}^n$, i.e. a set of points in $\mathbb{R}^n$ with certain structures. One can also define lattices over finite fields, rings or modules as used in many modern post-quantum crypto systems such as the CRYSTALS suites, NTRU and Saber.

Two problems form the very basis for computationally hard problems on lattices, namely the closest vector problem (CVP) and the shortest vector problem (SVP). Given a finite set of basis vectors in $\mathbb{R}^n$, the set of all linear combinations with integer coefficients forms a lattice. In optimization form, the SVP asks for the shortest vector in the lattice and the CVP asks for the lattice vector closest to some given target vector, both with respect to some given norm.

When working over the reals, the p-norm (for $p \geq 1$) is defined as $\sqrt[p]{\sum_i |x_i|^p}$. The most common examples are the Euclidean norm $\|x\|_2$ and the infinity norm $\|x\|_\infty = \max_i\{|x_i|\}$, which is the limit for $p \to \infty$.

We have formalized, corrected and verified a number of NP-hardness proofs from the literature, uncovering a number of mistakes along the way. The first NP-hardness proof of the CVP and SVP in infinity norm is due to van Emde-Boas [7]. For other norms (especially for the Euclidean norm), there is only a randomized reduction for the NP-hardness of the SVP so far [2]. For the CVP,

This work was supported by the Research Training Group GRK 2428 CONVEY of the German Research Council (DFG).

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 365–381, 2023.
https://doi.org/10.1007/978-3-031-38499-8_21

NP-hardness has been shown in any p-norm for $p \geq 1$. One exemplary proof can be found in the book by Micciancio and Goldwasser [15, Chapter 3, Thm 3.1].

The CVP and SVP were the starting point for lattice-based post-quantum cryptography [16]. Moreover, the relevance of these problems can also be seen from the rich literature on approximation results. For example, the LLL-algorithm by Lenstra, Lenstra and Lovász [12] gives a polynomial-time algorithm for lattice basis reduction which solves integer linear programs in fixed dimensions. Using this reduced basis, one can find good approximations to the CVP using Babai's algorithm [3] for certain approximation factors. Still, for arbitrary dimensions, the problem remains NP-hard. Further approximation results for the CVP, SVP and integer programming can be found elsewhere [6,9,10,14,19]. These approximation problems are used in cryptography. However, we will focus on the exact CVP and SVP in this paper.

A number of more basic NP-hardness proofs have been formalized in several theorem provers so far. For example, there are formalizations of the Cook-Levin Theorem in Coq [8] and Isabelle [4]. Formalizing Karp's 21 NP-hard problems (including the Subset Sum and Partition Problems assumed to be NP-hard in this paper) in Isabelle is an ongoing project.

1.1 Contributions

In this paper we present NP-hardness proofs of the CVP and SVP in infinity norm that have been verified in a proof assistant. We roughly follow the book by Micciancio and Golwasser [15, Chapter 3, Thm 3.1] and the report by van Emde-Boas [7]. However, many problems with the original proofs were encountered during the formalization efforts. We will have a look at different approaches and their advantages or problems.

We also verified the proof of NP-hardness of the CVP for any finite $p \geq 1$ from the book by Micciancio and Goldwasser. This verification did not uncover any problems with the informal proof. Thus we do not discuss it in detail.

These formalizations were carried out with the help of the proof assistant Isabelle [17,18] and are available online [11]. They comprise 5200 lines. To the authors knowledge, they are the first formalizations of hardness proofs for lattice problems. Because of the importance of the SVP and CVP and the problems in existing proofs, we consider our proofs a contribution to the foundations of verified cryptography. However, we do not claim that these hardness results directly imply quantum-resistance of any lattice-based cryptosystems.

1.2 Overview

The paper is structured as follows. Section 2 introduces the foundations. The rest of the paper is dedicated to the proofs, which are phrased as the following two polynomial time reduction chains:

- Subset Sum $\leq_p$ CVP
- Partition $\leq_p$ Bounded Homogeneous Linear Equations $\leq_p$ SVP

Subset Sum and Partition are famous fundamental problems whose NP-hardness has been proved many times in the literature and which we take for granted.

Section 3 presents the reduction of Subset Sum to the CVP. Differences between our formalization and the book by Micciancio and Goldwasser [15] are presented with examples that demonstrate problems with the original proof. Moreover, an example is given why the generalization to the SVP given in [15] does not work.

Therefore we turn to the early proof of NP-hardness of the SVP by van Emde Boas [7]. This proof uses the Bounded Homogeneous Linear Equations problem (BHLE) which is introduced in Sect. 4. The formalization of this proof is one of the major achievements in this paper. It posed a significant challenge since it often relied on human intuition and had to be restructured appropriately to allow a formal proof. The main proof steps are explained and difficulties in the formalization effort are described. This proof only works in infinity norm and we explain why. In Sect. 5, the reduction from BHLE to the SVP is given. Again, this proof was quite elaborate to formalize as there were inaccuracies and a lot of intuition was involved. Differences between the formal proof and [7] are explained by examples.

In Sect. 6, we have a quick look at the reduction proof for the CVP in p-norm (for finite $p \geq 1$). In the case of the SVP there only exists a randomized hardness proof in Euclidean norm by Ajtai [1] up to now.

Finally, the time complexity of the reduction functions are considered in Sect. 7. We conclude the paper with a short summary and outlook.

2 Foundations

This section introduces known foundations mainly to fix the terminology and notation: problem reductions, lattices, and the combinatorial problems under consideration (CVP, SVP, Partition and Subset Sum).

2.1 Problem Reductions

Formally, a *decision problem* is given by the set of *YES-instances* P and a set Γ of problem *instances*, where $P \subseteq \Gamma$. We often associate the decision problem with the set of YES-instances, when the instance set Γ is obvious and not explicitly defined. In this paper we will often phrase problems informally (e.g. "decide if p is prime") rather than give them explicitly as sets. For example, the decision problem "decide if a natural number p is prime" will be formalized in the following way: the set of problem instances is $\Gamma = \mathbb{N}$ (in Isabelle these are all elements of type *nat*); and the YES-instances are $P = \{p \in \mathbb{N} \mid p \text{ is prime}\}$ (in Isabelle this is a set of type *nat set*).

Definition 1 (Problem reduction). *Let $A \subseteq \Gamma$ and $B \subseteq \Delta$ be two problems. A function $f : \Gamma \to \Delta$ is a reduction from A to B if it fulfills the following properties:*

- $\forall a \in \Gamma.\ a \in A \Leftrightarrow f(a) \in B$
- *f can be computed in polynomial time*

If A is NP-hard, a reduction to B proves NP-hardness of B.

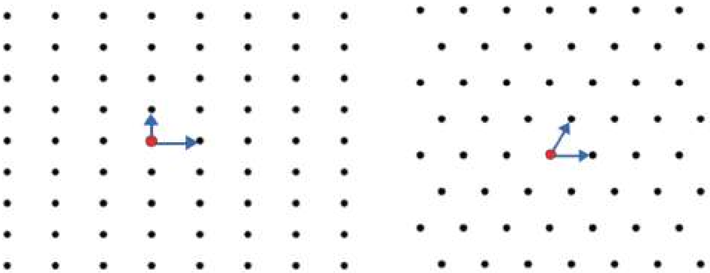
In this paper we present reduction functions informally (e.g. "an a is reduced to a b that is constructed like this") and often with copious amounts of "$\ldots$" to construct vectors etc. Of course in the formalization these reduction functions are spelled out in complete detail. Since all operations used in the reduction functions in this paper are elementary, the polynomial time property has not been formalized but is briefly discussed in Sect. 7. The focus of our paper are the proofs $a \in A \Leftrightarrow f(a) \in B$.

2.2 Lattice-Based Computational Problems

To have a better understanding, we will first introduce lattices as such. Lattices are a structured set of points. They form an additive, discrete subgroup of $\mathbb{R}^n$. Formally, we define the following.

Definition 2 (Lattice). *Let $A = \{a_1, \ldots, a_n\} \subset \mathbb{R}^n$ be a set of linearly independent vectors. Then the integer span of A forms a lattice $\mathcal{L}$, that is:*

$$\mathcal{L} = \left\{ \sum_{i=1}^{n} c_i a_i \mid c_i \in \mathbb{Z} \right\}$$



(a) Lattice with rectangular basis vectors

(b) Lattice with triangular basis vectors

Fig. 1. Two exemplary lattices in $\mathbb{R}^2$

Example 1. In Fig. 1 two examples of lattices in $\mathbb{R}^2$ are depicted. The red point is the origin. The two blue arrows show the basis vectors a_1 and a_2 that are linearly independent and span the lattice. Every integer combination of the two blue arrows is a black point, an element of the lattice.

We can see that the grid spanned by the basis vectors is discrete and has some recurring structures. These structures are determined by the basis vectors: the angle between them and their length. In Fig. 1a, the angle between the two basis vectors is 90° yielding a rectangular fundamental domain. Whereas in Fig. 1b, we have an angle of 60° between the basis vectors and equal length. This produces a fundamental domain of an equilateral triangle.

Indeed, the automorphism group of a lattice is a symmetry group, see Conway [5, Chapter 3.4]. For example, in Fig. 1a the symmetry group is **pmm** and in Fig. 1b is it **p3m1** [13].

In the rest of the text and in the formalization we restrict to finite bases over $\mathbb{Z}$ (instead of $\mathbb{R}$), simply for computability reasons. Of course bases over $\mathbb{Q}$ can be transformed into bases over $\mathbb{Z}$ by scaling all basis vectors.

The starting point of most known hard problems on lattices are the shortest vector problem and the closest vector problem. They are defined below (as usual in decision and not in optimization form). The lattice $\mathcal{L} \subseteq \mathbb{Z}^n$ is assumed to be generated by a finite basis in $\mathbb{Z}^n$.

Definition 3 (Closest Vector Problem (CVP)). *Given a lattice $\mathcal{L}$, a vector $b \in \mathbb{Z}^n$ and an estimate k, decide whether there exists a vector $v \in \mathcal{L}$ such that*

$$\|v - b\| \leq k$$

Definition 4 (Shortest Vector Problem (SVP)). *Given a lattice $\mathcal{L}$ and an estimate k, determine whether there exists a vector $v \in \mathcal{L}$ such that*

$$\|v\| \leq k \textit{ and } v \neq 0$$

2.3 Partition and Subset Sum Problems

Recall that we plan to prove NP-hardness of the CVP and SVP in the case of the infinity norm by reducing the well-studied NP-complete Subset Sum and Partition problems to the CVP and SVP. We state the definitions.

Definition 5 (Partition problem). *Given a finite list of integers $a_1, \dots, a_n$, does there exist a partition of $\{1 \dots n\}$ into subsets I and $\{1 \dots n\} \setminus I$ such that*

$$\sum_{i \in I} a_i = \sum_{i \in \{1 \dots n\} \setminus I} a_i$$

The Partition problem can be seen as a special case of the Subset Sum problem.

Definition 6 (Subset Sum problem). *Given a finite list of integers $a_1, \dots, a_n$ and an integer s, decide whether there exists a subset S of $\{1 \dots n\}$ such that*

$$\sum_{i \in S} a_i = s$$

2.4 Notation

Throughout the paper we use traditional mathematical notation, in particular the graphical "...". The formal Isabelle notation is by necessity more verbose (and precise). Our formalization employs both lists and vectors as a type for finite sequences and converts between them where necessary. For reasons of presentation we blur this distinction in the paper.

3 CVP

In this section, we formalize the proof of the NP-hardness of the CVP in the infinity norm along the lines of [15, p 48., Chap. 3.2, Thm 3.1] by reducing Subset Sum to the CVP.

An instance $a_1, \ldots, a_n, s$ of Subset Sum is mapped to the following instance of the CVP:

$$\mathcal{L} = \begin{pmatrix} a_1 & \cdots & a_n \\ a_1 & \cdots & a_n \\ 2 & & 0 \\ & \ddots & \\ 0 & & 2 \end{pmatrix} \cdot \mathbb{Z}^n \qquad b = \begin{pmatrix} s-1 \\ s+1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \qquad k = 1 \tag{1}$$

We proved the following theorem:

Theorem 1. *The above mapping is a reduction from the Subset Sum problem to the CVP (in infinity norm).*

This implies that the CVP (in infinity norm) is an NP-hard problem.

The reduction function used by Micciancio and Goldwasser [15] actually looks a bit different. The image of $a_1, \ldots, a_n, s$ would be

$$B = \begin{pmatrix} a_1 & \cdots & a_n \\ 2 & & 0 \\ & \ddots & \\ 0 & & 2 \end{pmatrix} \qquad \mathcal{L} = B \cdot \mathbb{Z}^n \qquad b = \begin{pmatrix} s \\ 1 \\ \vdots \\ 1 \end{pmatrix} \qquad k = 1 \tag{2}$$

However, the proof in [15, p. 49] with this reduction function works only for $p < \infty$. It goes along the lines of the following idea: Take $k = \sqrt[p]{n}$. In the case of $p = \infty$, we get $k = \lim_{p \to \infty} \sqrt[p]{n} = 1$. Then we can formulate the following equality (equation (3.5) in [15, p. 49]):

$$\|Bx - b\|_p^p = \left| \sum_{i=1}^{n} a_i x_i - s \right|^p + \sum_{i=1}^{n} |2x_i - 1|^p \tag{3}$$

Given a YES-instance $a_1, \ldots, a_n, s$ of Subset Sum, there exists a vector $x = (x_1, \ldots, x_n) \in \{0,1\}^n$, such that $\sum_{i=1}^{n} a_i x_i - s = 0$ and $|2x_i - 1| = 1$. Then $\|Bx - b\|_p^p = n$ which proves this case.

Given a YES-instance of the CVP defined by $\mathcal{L}$, t and k that are the image of $a_1, \dots, a_n, s$ under the reduction function as in (2), we get $\|Bx - b\|_p^p \leq n$. Since all values are integers, we have $|2x_i - 1| \geq 1$. It follows that $\sum_{i=1}^n a_i x_i - s = 0$ and $|2x_i - 1| = 1$. Thus, we can deduce that $a_1, \dots, a_n, s$ was indeed a YES-instance of Subset Sum.

The major problem we encountered was that this proof works fine for $p < \infty$ but for $p = \infty$, the sum in (3) becomes a maximum instead. The equation then reads

$$\|Bx - b\|_\infty = \max\left(\left|\sum_{i=1}^n a_i x_i - s\right|, |2x_i - 1| \text{ for } 1 \leq i \leq n\right)$$

This invalidates the arguments in the proof since $|\sum_{i=1}^n a_i x_i - s|$ can now be in the range $\{-1, 0, 1\}$. The constraints are too lax to ensure the equality to zero.

A solution was to alter the matrix and target vector and add another entry. The matrix and target vector we used are given in Eq. (1). The alternation to $s - 1$ and $s + 1$ forces a linear combination of the a_i to be exactly s in the hardness proof, since $|\sum_i c_i a_i - (s \pm 1)| \leq 1$.

After communicating with Daniele Micciancio, one of the authors of [15], he suggested using a constant $c > 1$ and the generating instance

$$\mathcal{L} = \begin{pmatrix} c \cdot a_1 & \cdots & c \cdot a_n \\ 2 & & 0 \\ & \ddots & \\ 0 & & 2 \end{pmatrix} \cdot \mathbb{Z}^n \qquad b = \begin{pmatrix} c \cdot s \\ 1 \\ \vdots \\ 1 \end{pmatrix} \qquad k = 1$$

This solves the problem as well and can be implemented using e.g. $c = 2$. This technique is described later in the book [15, pp. 49–51] when trying to explain the NP-hardness proof for the SVP in the infinity norm.

3.1 Towards the SVP

The authors of [15] argue that the reduction argument of the SVP can be deduced generating an instance of the SVP using the Subset Sum instance $a_1, \dots, a_n, s$ in the following way. For $c > 1$, e.g. $c = 2$, take

$$B = \begin{pmatrix} c \cdot a_1 & \cdots & c \cdot a_n & c \cdot s \\ 2 & & 0 & 1 \\ & \ddots & & 1 \\ 0 & & 2 & 1 \end{pmatrix} \qquad \mathcal{L} = B \cdot \mathbb{Z}^{n+1} \qquad k = 1$$

The authors claim that every shortest vector in the image of the reduction function has -1 as last coefficient. For example, let a YES-instance of the SVP be defined by the generating matrix B of the lattice and let $x = (x_1, \dots, x_n, -1)^T$

be the coefficients such that Bx is a shortest vector. Then we know that

$$\|Bx\|_\infty = \left\| \begin{pmatrix} c \cdot (x_1 a_1 + \cdots + x_n a_n - s) \\ 2x_1 - 1 \\ \vdots \\ 2x_n - 1 \end{pmatrix} \right\|_\infty \leq 1$$

Since $c > 1$, it follows, that $x_1 a_1 + \cdots + x_n a_n - s = 0$, which yields a solution for the given Subset Sum instance $a_1, \ldots, a_n, s$.

However, this reduction does not always work as the following example shows:

Example 2. Given the Subset Sum instance $(a_1, a_2, a_3, s) = (1, 1, 1, 1)$. This is a YES-instance, since a solution is given by $x_1 = 1$, $x_2 = 0$ and $x_3 = 0$. The basis matrix of the corresponding SVP would be (with $c > 1$)

$$B = \begin{pmatrix} c & c & c & c \\ 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 2 & 1 \end{pmatrix}$$

Take for example the vector $v = B \cdot (-1, -1, -1, 3)^T = (0, 1, 1, 1)^T$. It has infinity norm 1 and is thus a shortest vector in the lattice generated by B. However, this vector has the last coefficient 3 and not -1, even though it clearly is a shortest vector of the lattice given by B. The corresponding scaled "solution" for Subset Sum would be $(1/3, 1/3, 1/3, -1)$ but since only integer values are allowed in the solution space, this is not a solution in our sense.

We consider another example. Let the Subset Sum instance be $a'_1 = 3, s' = 1$. We can easily see that this is not a YES-instance, i.e. there exists no solution. Still, the corresponding SVP instance given via the reduction function is generated by the matrix

$$B' = \begin{pmatrix} c \cdot 3 & c \cdot 1 \\ 2 & 1 \end{pmatrix}$$

In this case the coefficients $(-1, 3)^T$ yield a shortest vector in the lattice spanned by B', since

$$\left\| B' \begin{pmatrix} -1 \\ 3 \end{pmatrix} \right\|_\infty = \left\| \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\|_\infty \leq 1$$

Thus, B' defines a YES-instance of the SVP, but the original Subset Sum instance is not a YES-instance.

In [15], it is stated for the infinity norm that any shortest vector yields a solution for the Subset Sum Problem, which is not the case in these examples: we cannot ensure that a shortest vector always has -1 as a last coordinate.

Although the proof in [15] does not work out as expected, there is still the reduction proof by van Emde-Boas [7] which reduces a problem called the Bounded Homogeneous Linear Equation problem to the SVP in infinity norm. This will be discussed in the next two sections.

4 Bounded Homogeneous Linear Equations

A technical report by Peter van Emde-Boas [7] gives another reduction proof for the NP-hardness of the SVP in infinity norm. The author first reduces the Partition Problem to a problem called Bounded Homogeneous Linear Equation (BHLE) which is then reduced to the SVP.

Definition 7 (Bounded Homogeneous Linear Equations problem). *Given a finite vector of integers $b \in \mathbb{Z}^n$ and a positive integer k, decide whether there exists an $x \in \mathbb{Z}^n \setminus \{0\}$ with $\|x\|_\infty \leq k$ such that*

$$\langle b, x \rangle = 0$$

We have verified a reduction from Partition to BHLE, and thus BHLE is NP-hard.

Theorem 2. *There is a reduction from Partition to BHLE in infinity norm.*

The proof is carefully engineered and rather intricate. Differences to the original proof and problems encountered during the formalization are:

- Our formal proof has a different structure than the proof in the technical report [7]. Indeed, the technical report first proves the reduction of a weaker form of Partition to BHLE and then argues that "omitting" an element yields the desired result as it adds stricter constraints. In the formalization we skip this intermediate step and directly prove the existence of an appropriate reduction function.
- Steps that seem trivial in the technical report often require a long formal proof. What can be reasoned by intuition in a pen-and-paper proof has to be elaborated in the formal proof. Intuition is also sometimes used for hand-waving over small gaps or imprecisions.
- Indexing vectors and lists has been a problem in the formalization. In pen-and-paper proofs, one can argue easily about "omitting" an element of a list even though this is imprecise and often misuses the notation. In the formalization one cannot simply skip an index. All indexing functions in the formalization have to be total. "Omitting" an element can only be solved by re-indexing and re-structuring the lists in the proof.
- Numbers are interpreted in different number systems during the proof. In contrast to the original proof, the formalization has to explicitly state the digits for a change of basis and show equivalence. This leads to verbose and elaborate proofs. To make proofs easier, we use the concrete basis $d = 5$ instead of an unspecified basis $d > 4$ as in [7]. Furthermore, the number M must use the absolute values of the a_i (omission in the definition of M in [7]). The formal definition is stated below.
- The proof involved many arguments about manipulations of huge sums. Working with huge sums entails very large proof states where the existing proof automation mostly failed on. These proof states require detailed (but still readable) proofs and occasional manual instantiation of theorems. Another possible solution to get smaller proof states is to introduce local abbreviations for subterms.

Let us have a look at the proof and its difficulties in the formalization in more detail. We start from a Partition instance $a = a_1, \ldots, a_n$. Note that we ignore the trivial case $n = 0$ in this presentation (but deal with it in the formal proofs)—this means $n - 1 \geq 0$. We reduce a to a BHLE instance b as follows:

- Define

$$M = 2 \cdot (\sum_{i=1}^{n} |a_i|) + 1 \tag{4}$$

- For $1 \leq i < n$ generate a 5-tuple

$$\begin{aligned} b_{i,1} &= a_i + M \cdot (5^{4i-4} + 5^{4i-3} + 5^{4i-1}) \qquad (5)\\ b_{i,2} &= M \cdot (5^{4i-3} + 5^{4i}) \\ b_{i,3} &= M \cdot (5^{4i-4} + 5^{4i-2}) \\ b_{i,4} &= a_i + M \cdot (5^{4i-2} + 5^{4i-1} + 5^{4i}) \\ b_{i,5} &= M \cdot (5^{4i-1}) \\ b_i &= b_{i,1}, b_{i,2}, b_{i,4}, b_{i,5}, b_{i,3} \end{aligned}$$

  Note that $b_{i,3}$ has moved to the last position in b_i.
- For $i = n$ generate only a 4-tuple:

$$\begin{aligned} b_{n,1} &= a_n + M \cdot (5^{4n-4} + 5^{4n-3} + 5^{4n-1}) \\ b_{n,2} &= M \cdot (5^{4n-3} + 1) \\ b_{n,4} &= a_n + M \cdot (5^{4n-2} + 5^{4n-1} + 1) \\ b_{n,5} &= M \cdot (5^{4n-1}) \qquad (6)\\ b_n &= b_{n,1}, b_{n,2}, b_{n,4}, b_{n,5} \end{aligned}$$

  Note that
  - $b_{n,3}$ is omitted from b_n to restrict the constraints necessary for the proof and
  - that in $b_{n,2}$ and $b_{n,4}$ the last summand changes to a $+1$ in comparison to the other $b_{i,2}$ and $b_{i,4}$.

In summary, the entry $b_{i,3}$ is uniformly in the last position in the b_i but omitted from the final b_n.

The Partition instance a of length n is reduced to a vector b of length $5n - 1$:

$$b = (b_1, \ldots, b_{n-1}, b_n) \tag{7}$$

The NP-hardness proof now follows in three steps:

1. We need to show an auxiliary lemma.
2. We show that a YES-instance of Partition is reduced to a YES-instance of BHLE.
3. We show that the pre-image of a YES-instance of BHLE is indeed a YES-instance in Partition.

4.1 Auxiliary Lemma

As a first step, the proof needs a short auxiliary lemma from number theory.

Lemma 1. *Let $x, y, c \in \mathbb{Z}^n$ and M be an integer. Assume that $M > \sum_{i=1}^{n} |x_i|$ and that $|c_i| \leq 1$ for all $1 \leq i \leq n$. Furthermore, let the following equation hold:*

$$\sum_{i=1}^{n} c_i \cdot (x_i + M \cdot y_i) = 0 \tag{8}$$

Then we have

$$\langle c, x \rangle = 0 \quad \text{and} \quad \langle c, y \rangle = 0$$

In this lemma, we can reinterpret $x_i + M \cdot y_i$ from (8) as a number in basis M with lowest digit x_i. Even with a coefficient c_i, the lowest digit in basis M has to be zero, as well as the rest. By splitting off the lowest digits consecutively, we can show, that indeed all digits in basis M have to equal zero.

4.2 $a \in$ Partition $\Longrightarrow b \in$ BHLE

This direction is quite easy. Let $a_1, \ldots, a_n$ be a YES-instance of partition with partitioning set I. We will show that the following vector x is a solution to the corresponding BHLE:

$$\begin{aligned}
x &= (x_1, \ldots, x_{n-1}, x_n) \\
x_i &= \begin{cases} 1, -1, 0, -1, 0 & i \in I \wedge n-1 \in I \\ 0, 0, -1, 1, 1 & i \in I \wedge n-1 \notin I \\ 0, 0, -1, 1, 1 & i \notin I \wedge n-1 \in I \\ 1, -1, 0, -1, 0 & i \notin I \wedge n-1 \notin I \end{cases} \qquad 1 \leq i < n \\
x_n &= 1, -1, 0, -1
\end{aligned}$$

We have to show that $\langle b, x \rangle = 0$. This is proven by plugging in the definitions and rearranging terms in the sum of the scalar product such that they cancel out. As a last step in the proof, we need to show that $\|x\|_\infty \leq 1$. For the infinity norm this is quite easy. However, it would not be true for other norms. For $p \geq 1$ and $p < \infty$ we have for $n \geq 1$:

$$\|x\|_p = \sqrt[p]{3n} > 1$$

Thus, the chosen constraints x only work in infinity norm.

4.3 $a \in$ Partition $\Longleftarrow b \in$ BHLE

This direction is harder. Let b be a YES-instance of BHLE. That is, there exists a nonzero x such that $\langle b, x \rangle = 0$ and $\|x\|_\infty \leq 1$. We have to show that there is a partition I on $a_1, \ldots, a_n$ with $\sum_{i \in I} a_i = \sum_{i \in \{1 \ldots n\} \setminus I} a_i$.

The proof idea works as follows. First, we apply the auxiliary lemma and get a constraint on the a_i on the one hand, and a condition on the x_i with coefficients that are powers of 5 on the other hand. Using this condition on the x_i, we generate equational constraints on the entries of x by looking at the digits in basis 5. We argue that a number equals zero if and only if all its digits are zero.

The generated equations lead to a good characterisation of x, namely the weight $w = x_{5(n-1)+1}$. From the assumption that $\|x\|_\infty \leq 1$, we deduce $|w| \leq 1$. Again, this step can only be reasoned in the infinity norm. For other p-norms, this argumentation breaks as we need the property $|w| \leq 1$ to complete the proof. Using the value of w, we can constuct a partitioning set I with the required property from the equation on the a_i.

5 SVP

Knowing that the BHLE is indeed an NP-hard problem, we reduce it to the SVP. Then we can conclude that the SVP in infinity norm is NP-hard.

Theorem 3. *There is a reduction from BHLE to the SVP in infinity norm.*

Again some difficulties were met when formalizing the proof for the above theorem. First of all, note that the terminology in [7] and nowadays is a bit different. In [7], the shortest vector problem only denotes the shortest vector problem in the Euclidean norm. What we call the shortest vector problem in the infinity norm is named closest vector problem in [7]. To make terminology even more confusing, our understanding of the closest vector problem is called the nearest vector problem in [7]. To make the notation clear, we provide a table for reference in Fig. 2.

technical report [7]	our notation
closest vector problem	SVP in infinity norm
shortest vector problem	SVP in Euclidean norm
nearest vector problem	CVP

Fig. 2. Notation

A more mathematical problem encountered was that the reduction itself used in [7] was not entirely correct. In the reduction two factors $k' = k+1$ and k'' were introduced. These factors should have certain properties to allow the arguments of the reduction proof to go through. However, this is only true when tweaking these factors a bit to make the whole proof watertight. We will now have a closer look.

Given the BHLE instance $b = (b_1, \ldots, b_n)$ and k, create the following SVP instance:

$$\mathcal{L} = \begin{pmatrix} 1 & & 0 & 0 \\ & \ddots & & \vdots \\ 0 & & 1 & 0 \\ - & (k+1) \cdot b & - & k'' \end{pmatrix} \cdot \mathbb{Z}^n \qquad k = k$$

where k'' is the factor in question. In the technical report, we have

$$k'' = 2 \cdot (k+1) \cdot (\sum_i b_i) + 1$$

The following example however shows that this factor is not enough.

Example 3. Consider the BHLE instance given by $b = (1, -1)$ and $k = 1$. This is a YES-instance, since the vector $(1, 1)$ yields the expected properties.

Define the following matrices.

$$B_0 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & -2 & 1 \end{pmatrix} \qquad B_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & -2 & 9 \end{pmatrix} \qquad B_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 6 & -6 & 25 \end{pmatrix}$$

The associated SVP instance is the lattice generated by B_0. Then the vector $(0, 0, 1)^T$ with infinity norm 1 is a solution to the SVP instance generated by the basis matrix B_0. However, since the last entry is nonzero, this does not provide a solution for BHLE. Contrary to this example, the proof in the technical report shows that for all SVP solutions the last entry must be zero.

The reason, why the argument in the technical report breaks at this point is because $b_1 + b_2 = 0$, thus making $k'' = 1$ very small. One step to prevent this is to use the absolute values of the b_i in k'' instead. The new k''_1 we consider is

$$k''_1 = 2 \cdot (k+1) \cdot (\sum_i |b_i|) + 1$$

With this new factor k''_1 we get the generating matrix B_1 and the vector $(0, 0, 1)$ is no longer a shortest vector.

Still, this is not enough. Consider the same $b = (1, -1)$ as above, but let $k = 5$. Then we get B_2 as the generating matrix of the SVP lattice. The vector $x = (0, 5, 1)^T$ is a shortest vector whose last entry is nonzero. Again it contradicts the proof in the technical report. The reason this time is the following: the argument that $(k+1)\,(\sum_{i=1}^n x_i b_i)$ and k''_1 have different relative sizes fails. Indeed, we have

$$\left\| \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 6 & -6 & 25 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 5 \\ 1 \end{pmatrix} \right\|_\infty = \left\| \begin{pmatrix} 0 \\ 5 \\ -5 \end{pmatrix} \right\|_\infty = 5 \leq k$$

We can obtain different relative sizes of $(k+1)\,(\sum_{i=1}^n x_i b_i)$ and k''_1 by defining

$$k''_2 = 2 \cdot k \cdot (k+1) \cdot (\sum_i |b_i|) + 1 \tag{9}$$

Now we can make sure that the last entry of a solution to the SVP problem is indeed zero. For the proof of Theorem 3 we consider the reduction given by

$$\mathcal{L} = \underbrace{\begin{pmatrix} 1 & & 0 & 0 \\ & \ddots & & \vdots \\ 0 & & 1 & 0 \\ \multicolumn{4}{c}{-\ (k+1)\cdot b - k_2''} \end{pmatrix}}_{B} \cdot \mathbb{Z}^n \qquad k = k$$

where B denotes the basis matrix generating the lattice $\mathcal{L}$ as given above.

Consider a solution $x = (x_1, \ldots, x_{n+1})$ of the SVP with $\|Bx\|_\infty \leq k$. Then we have

$$Bx = \begin{pmatrix} 1 & & 0 & 0 \\ & \ddots & & \vdots \\ 0 & & 1 & 0 \\ \multicolumn{4}{c}{-\ (k+1)\cdot b - k_2''} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ x_{n+1} \end{pmatrix} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ (k+1)(\sum_{i=1}^{n} x_i b_i) + x_{n+1} \cdot k_2'' \end{pmatrix}$$

As this yields a solution to the SVP, we get:

$$|(k+1)(\sum_{i=1}^{n} x_i b_i) + x_{n+1} \cdot k_2''| \leq k \tag{10}$$

Then we calculate:

$$\begin{aligned} (k+1)(\sum_{i=1}^{n} x_i b_i) + x_{n+1} \cdot k_2'' \leq (k+1)(\sum_{i=1}^{n} |x_i||b_i|) + x_{n+1} \cdot k_2'' \leq \\ \leq (k+1)k(\sum_{i=1}^{n} |b_i|) + x_{n+1} \cdot k_2'' \end{aligned}$$

Assuming that $x_{n+1} \neq 0$, we have

$$|(k+1)k(\sum_{i=1}^{n} |b_i|)| < |2 \cdot k \cdot (k+1) \cdot (\sum_{i} |b_i|) + 1| = |k_2''| \leq |x_{n+1} \cdot k_2''|$$

Thus the two summands indeed have different relative sizes and can never cancel out the other summand. This leads to a contradiction to (10). Therefore, $x_{n+1} = 0$ must be true and $(x_1, \ldots, x_n)$ constitutes a solution to the BHLE when using k_2'' as in (9).

6 Other p-Norms

Up to now, we have investigated lattice problems under the infinity norm. Even though this yields nice hardness results, in practice the Euclidean norm is used more often. Unfortunately, when considering p-norms things do not play out as nicely. In this section, we assume $1 \leq p < \infty$ whenever we talk about a specific p.

For the CVP, there is a generalisation of the proof for every p-norm in [15, p. 48, Chap. 3.2, Thm 3.1] which we also formalized. Let $a_1, \ldots, a_n, s$ be an instance of Subset Sum. The reduction function maps this instance to:

$$\mathcal{L} = \begin{pmatrix} a_1 & \cdots & a_n \\ 2 & & 0 \\ & \ddots & \\ 0 & & 2 \end{pmatrix} \cdot \mathbb{Z}^n \qquad b = \begin{pmatrix} s \\ 1 \\ \vdots \\ 1 \end{pmatrix} \qquad k = \sqrt[p]{n}$$

Then the following theorem holds:

Theorem 4. *The above mapping is a reduction from the Subset Sum problem to the CVP in p-norm.*

This implies that the CVP in p-norm is an NP-hard problem. The outline to the proof is given in Sect. 3 after Theorem 1. The important difference to the infinity norm is that the bound k scales with the dimension n of the lattice.

For the SVP, there is no known deterministic NP-hardness result in the Euclidean norm, or even any p-norm. However, Ajtai [1,2] found an interesting alternative which is quite useful for the application in cryptography, namely randomized reductions using polynomial-time probabilistic reduction functions. In cryptography, these results guarantee the hardness of "average" cases. That is, given an average instance according to a probability distribution, it will most likely be intractable.

7 Time Complexity

As stated in Sect. 2, time complexity of the above reduction functions has not been formalized. However, we give a short explanation why all reduction functions are indeed in polynomial time.

Subset Sum to CVP: The reduction function as given in Eq. (1) creates $(n+2)(n+1)+1$ values using only memory access or one addition. Therefore, the time complexity in this case is $\mathcal{O}(n^2)$.

Partition to BHLE: In this case, the reduction function maps the input a of length n to b as defined in Eq. (7). The value $k = 1$ is fixed. Then a is mapped to a vector of length $5n - 1$. When calculating the b_i, we need to calculate the value of M as in (4). As we sum over all input values, this lies in $\mathcal{O}(n)$. Each b_i can then be calculated in $\mathcal{O}(n)$ since it only contains a constant number of additions of the input with fixed cofactors (see (5)–(6)). Putting the construction of the list and the calculation of the b_i together, we find that the whole reduction function is in $\mathcal{O}(n^2)$.

BHLE to the SVP: Consider the reduction function as given in Eq. (5) using the value k_2'' as in (9). Calculating k_2'' requires $n+2$ memory accesses which are processed in $n+4$ arithmetic operations, thus having a time complexity of $\mathcal{O}(n)$. Every other entry in the matrix is calculated on $\mathcal{O}(1)$, since they contain at most two memory accesses and at most two arithmetic operations. The input generates $(n+1)^2+1$ values, of which $(n+1)(n+1)$ are in $\mathcal{O}(1)$ (namely all the zeros and ones, the vector $(k+1)\cdot a$ and the constraint k) and one is calculated in $\mathcal{O}(n)$ (namely k_2''). Thus, the whole reduction function lies in $\mathcal{O}(n^2)$.

8 Outlook

With this paper, we now have a formal proof for NP-hardness of the CVP and SVP in the infinity norm, as well as a formal proof of the CVP in p-norm (for $1 \leq p < \infty$). In the formalization process, many gaps and imprecisions in the pen-and-paper proofs were fixed. The changes to the original proofs have been elaborated with explanations and examples. Unfortunately, giving a deterministic reduction proof of the SVP in p norm for $p < \infty$ is still an open problem. Under probabilistic assumptions, Ajtai showed NP-hardness of the SVP in Euclidean norm in [2].

An interesting topic for future work is to develop a framework for probabilistic reductions such as in [2]. This will give the foundation to extend formalization of hardness proofs to other problems in lattice theory, especially those used in lattice-based cryptography, such as the Learning with Errors (LWE) Problem, Ring-LWE and Module-LWE. This will underline the security of many lattice-based crypto systems. Another topic for future work is to formalize the hardness proofs for approximate versions of the CVP and SVP.

Acknowledgements. We thank Manuel Eberl for continuous support and fruitful discussions. The first author gratefully acknowledges the financial support of this work by the research training group ConVeY funded by the German Research Foundation under grant GRK 2428.

References

1. Ajtai, M.: Generating hard instances of lattice problems. Electron. Colloquium Comput. Complex. **3** (1996)
2. Ajtai, M.: The shortest vector problem in L2 is NP-hard for randomized reductions (extended abstract). In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, STOC 1998, Dallas, Texas, USA, pp. 10–19. ACM Press (1998)
3. Babai, L.: On Lovász' lattice reduction and the nearest lattice point problem. Combinatorica **6**, 1–13 (1986)
4. Balbach, F.J.: The Cook-Levin theorem. Archive of Formal Proofs (2023). https://isa-afp.org/entries/Cook_Levin.html. Formal proof development
5. Conway, J.H., Sloane, N.J.A.: Sphere Packings, Lattices and Groups. Springer, New York (1999). https://doi.org/10.1007/978-1-4757-6568-7
6. Dinur, I., Kindler, G., Raz, R., Safra, S.: Approximating CVP to within almost-polynomial factors is NP-hard. Combinatorica **23**, 205–243 (2003). https://doi.org/10.1007/s00493-003-0019-y
7. van Emde Boas, P.: Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Technical report 81-04. Technical report, Mathematisch Instituut, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands (1981)

8. Gäher, L., Kunze, F.: Mechanising complexity theory: the Cook-Levin theorem in coq. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPICS.ITP.2021.20. https://drops.dagstuhl.de/opus/volltexte/2021/13915/
9. Haviv, I., Regev, O.: Tensor-based hardness of the shortest vector problem to within almost polynomial factors. In: Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing, STOC 2007, pp. 469–477. Association for Computing Machinery, New York (2007)
10. Khot, S.: Hardness of approximating the shortest vector problem in lattices. J. ACM **52**(5), 789–808 (2005)
11. Kreuzer, K.: Hardness of lattice problems. Archive of Formal Proofs (2023). https://isa-afp.org/entries/CVP_Hardness.html. Formal proof development
12. Lenstra, A.K., Lenstra, H., Lovasz, L.: Factoring polynomials with rational coefficients. Math. Ann. **261**, 515–534 (1982)
13. Liu, Y., Collins, R.: Frieze and wallpaper symmetry groups classification under affine and perspective distortion. Technical report. CMU-RI-TR-98-37, Carnegie Mellon University, Pittsburgh, PA (1998)
14. Micciancio, D.: The shortest vector in a lattice is hard to approximate to within some constant. In: Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No. 98CB36280), pp. 92–98 (1998). https://doi.org/10.1109/SFCS.1998.743432
15. Micciancio, D., Goldwasser, S.: Complexity of Lattice Problems. Springer, Boston (2002)
16. Micciancio, D., Regev, O.: Lattice-based cryptography. In: Bernstein, D.J., Buchmann, J., Dahmen, E. (eds.) Post-Quantum Cryptography, pp. 147–191. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-88702-7_5
17. Nipkow, T., Klein, G.: Concrete Semantics with Isabelle/HOL. Springer, Cham (2014). http://concrete-semantics.org
18. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL—A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
19. Rothvoss, T., Venzin, M.: Approximate CVP in time $2^{0.802n}$ – now in any norm! arXiv:2110.02387 [cs] (2021)

Buy One Get 14 Free: Evaluating Local Reductions for Modal Logic

Cláudia Nalon[1], Ullrich Hustadt[2](✉), Fabio Papacchini[3], and Clare Dixon[4]

[1] Department of Computer Science, University of Brasília, Brasília, Brazil
nalon@unb.br

[2] Department of Computer Science, University of Liverpool, Liverpool, UK
U.Hustadt@liverpool.ac.uk

[3] School of Computing and Communications, Lancaster University in Leipzig, Leipzig, Germany
f.papacchini@lancaster.ac.uk

[4] Department of Computer Science, University of Manchester, Manchester, UK
clare.dixon@manchester.ac.uk

Abstract. We are interested in widening the reasoning support for propositional modal logics in the so-called modal cube. The modal cube consists of extensions of the basic modal logic K with an arbitrary combination of the modal axioms B, D, T, 4 and 5. We revisit recently developed local reductions from all logics in the modal cube to a normal form comprising sets of clausal formulae with associated modal levels. We extend these reductions further to the basic modal logic K, called *definitional reductions*. This enables any prover for K to be used to solve the satisfiability problem for all logics in the modal cube. We also present alternative, *axiomatic*, reductions based on ideas originally proposed by Kracht, providing new theoretical results and improved bounds on the size of the reductions. We compare both sets of reductions combined with state-of-the-art provers for K on a large set of parametric benchmarks for all logics in the modal cube. The results show that the provers perform better with reductions based on the clausal normal form than the axiomatic reductions.

1 Introduction

Following [4], modal logics can be seen as simple but expressive languages for talking about relational structures that provide an internal and local perspective on those structures. The most intensively studied modal logics are the basic modal logic K and its extensions with one or more of the axioms B (symmetry), D (seriality), T (reflexivity), 4 (transitivity) and 5 (Euclideaness), that form

C. Nalon was partially supported by FAPDF 11/2021, DPG/UnB 004/2022. C. Dixon was partially supported by the EPSRC funded RAI Hubs FAIR-SPACE (EP/R026092/1) and RAIN (EP/R026084/1), and the EPSRC funded programme Grant S4 (EP/N007565/1).

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 382–400, 2023.
https://doi.org/10.1007/978-3-031-38499-8_22

the so-called *modal cube*. There are numerous reasons for this. To name just three: (i) relations which are serial, symmetric, transitive, etc. are very common; (ii) the logics in the modal cube can be used to represent and reason about idealised mental attitudes such as knowledge, belief, desire and intention; (iii) mathematical techniques, algorithms, calculi, as well as implemented reasoning tools for these logics provide building blocks for the study and application of more complex modal logics.

In [27], we have presented a reduction from each of the 15 distinct logics in the modal cube to Separated Normal Form with Sets of Modal Levels, SNF_{sml}, a clausal normal form for basic modal logic in which clauses are labelled with possibly infinite sets of modal levels, and to Separated Normal Form with Modal Levels, SNF_{ml}, where each clause is given a natural number label. The latter reduction then allowed us to use the modal-layered clausal resolution (MLR) calculus [22], implemented in the modal logic theorem prover K$_\mathsf{S}$P [19,26] to reason in these logics. We evaluated this approach on a new collection of benchmark formulae for all 15 logics and compared its performance with that of the global modal resolution (GMR) calculus also implemented in K$_\mathsf{S}$P and with Leo-III, an automated theorem prover for polymorphic higher-order logic [32]. The GMR calculus has specific rules for each logic while Leo-III reasons about modal logics using a translation approach and has translations for each of the 15 logics built in. The evaluation showed that the approach performs better than Leo-III but not as well as the GMR calculus in K$_\mathsf{S}$P. We identified the reduction from SNF_{sml} to SNF_{ml} as the main contributing factor, in particular, on satisfiable formulae where the MLR calculus has to fully saturate the corresponding set of SNF_{ml} clauses up to redundancy before it can conclude that the original formula is satisfiable.

In this paper, we investigate and evaluate an alternative use of our reductions from logics in the modal cube to SNF_{ml}. A finite set of clauses in SNF_{ml} can straightforwardly be transformed into a formula in the basic modal logic K. Such a transformation then allows the use of any existing approach to solving the satisfiability problem in K to the satisfiability problem in all logics in the modal cube. An advantage of the use of this transformation over a translation from each of the 15 logics to first-order (or higher-order) logic [1,5,9,14] is the availability of implemented decision procedures for basic modal logic. In contrast, while many decidable fragments of first-order logics are known, including decidable fragments that are suitable targets of translations of modal logic formulae, implemented decision procedures for these fragments are rare. See also related discussions in [27,30].

The original motivation for our work on reductions to SNF_{sml} and SNF_{ml} were Kracht's reductions of the normal modal logics KB, KD, KT, and $\mathsf{K4}$ to K [17,18]. Extending our reduction from SNF_{ml} to K to obtain a reduction from the modal cube to K raises first the question whether one can devise a reduction based on the same idea as Kracht's for the remaining logics of the modal cube. We will call such a reduction *axiomatic* as the idea is to use certain instances of axiom schemata embedded into modal contexts of nested $\Box$-operators up to a certain depth bound. We answer this question positively by providing the reductions missing in Kracht's work. The second question then raised is how well provers for K perform on our reduction compared to an axiomatic reduction. Our empirical evaluation indicates that the definitional reduction appears to result in better performance overall when combined with state-of-the-art K provers.

The structure of the paper is as follows. In Sect. 2 we recall common concepts of propositional modal logics and the definition of our normal form SNF_{ml}. Section 3 recalls our reduction from logics in the modal cube to SNF_{ml}, defines the transformation of a finite set of SNF_{ml} clauses to basic modal, and introduces the axiomatic reduction for the logics in the modal cube. In Sect. 4 we compare the performance of a combination of the reductions defined in Sect. 3 when combined with provers for basic modal logic as well as with the global resolution calculus for logics in the modal cube implemented in KSP.

2 Preliminaries

The language of modal logic is an extension of the language of propositional logic with unary modal operators $\Box$ and $\Diamond$. More precisely, given a denumerable set of *propositional symbols*, $P = \{p, p_0, q, q_0, t, t_0, \ldots\}$ as well as propositional *constants* **true** and **false**, *modal formulae* are inductively defined as follows: constants and propositional symbols are modal formulae. If φ and ψ are modal formulae, then so are $\neg\varphi$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$, $\Box\varphi$, and $\Diamond\varphi$. We also assume that $\wedge$, and $\vee$ are associative and commutative operators and consider, e.g., $(p \vee (q \vee r))$ and $(r \vee (q \vee p))$ to be identical formulae. We often omit parentheses if this does not cause confusion. The size of φ is the number of occurrences of propositional constants, propositional variable, boolean operators and modal operators in φ. By $\mathsf{var}(\varphi)$ we denote the set of all propositional symbols occurring in φ. This function easily extends to finite sets of modal formulae. A *modal axiom (schema)* is a modal formula ψ representing the set of all instances of ψ.

A *literal* is either a propositional symbol or its negation; the set of literals is denoted by L_P. By $\neg l$ we denote the *complement* of the literal $l \in L_P$, that is, if l is the propositional symbol p then $\neg l$ denotes $\neg p$, and if l is the literal $\neg p$ then $\neg l$ denotes p. By $|l|$ for $l \in L_P$ we denote p if $l = p$ or $l = \neg p$. A *modal literal* is either $\Box l$ or $\Diamond l$, where $l \in L_P$.

An occurrence of a subformula has *positive polarity* if it is inside the scope of an even number of (explicit or implicit) negations, and it has *negative polarity* if it is one inside the scope of an odd number of negations. A literal is *pure* if all its occurrences have either a positive or a negative polarity.

The modal logic K is given by the smallest set of modal formulae which includes all propositional tautologies, the axiom schema $\Box(\varphi \rightarrow \psi) \rightarrow (\Box\varphi \rightarrow \Box\psi)$, is closed under modus ponens and the rule of necessitation (if $\varphi \in \mathsf{K}$ then $\Box\varphi \in \mathsf{K}$). Given a modal logic L and set of axioms Σ, the smallest modal logic $L' \supset L \cup \Sigma$ is an *extension* of L and we denote L' by $L\Sigma$.

The standard semantics of modal logics is the *Kripke semantics* or *possible world semantics*. A *Kripke frame* F is an ordered pair $\langle W, R \rangle$ where W is a non-empty set of *worlds* and R is a binary (accessibility) relation over W. A *Kripke* *structure* M over P is an ordered pair $\langle F, V\rangle$ where F is a Kripke frame and the *valuation* V is a function mapping each propositional symbol in P to a subset $V(p)$ of W. A *rooted Kripke structure* is an ordered pair $\langle M, w_0\rangle$ with $w_0 \in W$.

Satisfaction (or truth) of a formula at a world w of a Kripke structure $M = \langle W, R, V\rangle$ is inductively defined by:

$$\begin{array}{ll}
\langle M, w\rangle \models \mathbf{true}; & \langle M, w\rangle \not\models \mathbf{false}; \\
\langle M, w\rangle \models p & \text{iff } w \in V(p), \text{ where } p \in P; \\
\langle M, w\rangle \models \neg\varphi & \text{iff } \langle M, w\rangle \not\models \varphi; \\
\langle M, w\rangle \models (\varphi \wedge \psi) & \text{iff } \langle M, w\rangle \models \varphi \text{ and } \langle M, w\rangle \models \psi; \\
\langle M, w\rangle \models (\varphi \vee \psi) & \text{iff } \langle M, w\rangle \models \varphi \text{ or } \langle M, w\rangle \models \psi; \\
\langle M, w\rangle \models (\varphi \rightarrow \psi) & \text{iff } \langle M, w\rangle \models \neg\varphi \text{ or } \langle M, w\rangle \models \psi; \\
\langle M, w\rangle \models \Box\varphi & \text{iff for every } v,\ w\,R\,v \text{ implies } \langle M, v\rangle \models \varphi; \\
\langle M, w\rangle \models \Diamond\varphi & \text{iff there is } v,\ w\,R\,v \text{ and } \langle M, v\rangle \models \varphi.
\end{array}$$

If $\langle M, w\rangle \models \varphi$ then we say that φ is true at w in M. A rooted Kripke structure $M = \langle M, w_0\rangle$ is a *model* of a modal formula φ iff $\langle M, w_0\rangle \models \varphi$ and M satisfies φ. A modal formula is *satisfiable* iff there exists a Kripke structure M and a world $w \in M$ such that $\langle M, w\rangle \models \varphi$. A rooted Kripke structure $M = \langle W, R, V, w_0\rangle$ is a *rooted tree Kripke structure* iff R is a tree, that is, a directed acyclic connected graph where each node has at most one predecessor, with *root* w_0.

A *path from* w'_0 *to* w'_k *of length* k, $k \geq 0$, in a frame $F = \langle W, R\rangle$ is a sequence $(w'_0, w'_1, \ldots, w'_k)$ where for every i, $0 \leq i \leq k-1$, $w'_i\,R\,w'_{i+1}$. A path (w'_0) of length 0 is identified with its root w'_0. In a rooted tree Kripke structure M with root w_0 for every world $w_k \in W$ there is exactly one path connecting w_0 and w_k; the *modal level* (in M), denoted by $\mathsf{ml}_M(w_k)$, is given by the length of the path from w_0 to w_k. More generally, for a rooted Kripke structure M with root w_0, the *depth* of a world w_k (in M), denoted by $\mathsf{depth}_M(w_k)$, is the length of the shortest path from w_0 to w_k. The *depth* of M is the maximal depth of a world in M. The *outdegree* of a world w in F is given by $|\{w' \mid w\,R\,w'\}|$.

The 15 logics in the modal cube consist of K itself and its extensions with one or more of the modal axioms shown in Table 1. Each of these axioms defines a class of Kripke frames where the accessibility relation R satisfies the first-order property stated in the table. Combinations Σ of axioms then define a class $\mathfrak{F}_\Sigma$ of Kripke frames where the accessibility relation satisfies the combination of their corresponding properties. Given a logic $L = \mathsf{K}\Sigma$, a modal formula φ is

Table 1. Modal axioms and relational frame properties

Name	Axiom	Frame Property	
D	$\Box\varphi \rightarrow \Diamond\varphi$	$\forall v \exists w. v\,R\,w$	Serial
T	$\Box\varphi \rightarrow \varphi$	$\forall w. w\,R\,w$	Reflexive
B	$\varphi \rightarrow \Box\Diamond\varphi$	$\forall vw. v\,R\,w \rightarrow w\,R\,v$	Symmetric
4	$\Box\varphi \rightarrow \Box\Box\varphi$	$\forall uvw. (u\,R\,v \wedge v\,R\,w) \rightarrow u\,R\,w$	Transitive
5	$\Diamond\varphi \rightarrow \Box\Diamond\varphi$	$\forall uvw. (u\,R\,v \wedge u\,R\,w) \rightarrow v\,R\,w$	Euclidean

Table 2. Rewriting Rules for Simplification

$\varphi \wedge \varphi \Rightarrow \varphi$	$\varphi \wedge \neg\varphi \Rightarrow \mathbf{false}$	$\Diamond\varphi \vee \Diamond\neg\varphi \Rightarrow \Diamond\mathbf{true}$
$\varphi \vee \varphi \Rightarrow \varphi$	$\varphi \vee \neg\varphi \Rightarrow \mathbf{true}$	$\Box\varphi \wedge \Box\neg\varphi \Rightarrow \Box\mathbf{false}$
$\neg\mathbf{true} \Rightarrow \mathbf{false}$	$\varphi \wedge \mathbf{true} \Rightarrow \varphi$	$\Box\mathbf{false} \wedge \Diamond\varphi \Rightarrow \mathbf{false}$
$\neg\mathbf{false} \Rightarrow \mathbf{true}$	$\varphi \wedge \mathbf{false} \Rightarrow \mathbf{false}$	$\Diamond\mathbf{true} \vee \Box\varphi \Rightarrow \mathbf{true}$
$\neg\neg\varphi \Rightarrow \varphi$	$\varphi \vee \mathbf{false} \Rightarrow \varphi$	$\Box\varphi \wedge \Diamond\neg\varphi \Rightarrow \mathbf{false}$
$\Box\mathbf{true} \Rightarrow \mathbf{true}$	$\varphi \vee \mathbf{true} \Rightarrow \mathbf{true}$	$\Box\mathbf{false} \wedge \Box\varphi \Rightarrow \Box\mathbf{false}$
$\Diamond\mathbf{false} \Rightarrow \mathbf{false}$		$\Diamond\mathbf{true} \vee \Diamond\varphi \Rightarrow \Diamond\mathbf{true}$

L-satisfiable iff there exists a frame $F \in \mathfrak{F}_{\Sigma}$, a valuation V and a world $w \in F$ such that $M = \langle F, V, w\rangle \models \varphi$ and we call M an L-model of φ.

A modal formula is in *simplified NNF* (denoted by $\mathsf{nnf}(\varphi)$), if it has been simplified by exhaustively applying the rewrite rules in Table 2, and it is in Negation Normal Form (NNF), that is, a formula where only propositional symbols are allowed in the scope of negations.

The reductions given in the next section produce formulae in a clausal normal form, called *Separated Normal Form with Sets of Modal Levels* SNF_{sml}, given in [29]. The language of SNF_{sml} extends that of the basic modal logic K with sets of modal levels as labels. Clauses in SNF_{sml} have one of the following forms:

$S : \bigvee_{i=1}^{n} l_i$	$S : l' \to \Box l$	$S : l' \to \Diamond l$
(literal clause)	(positive modal clause)	(negative modal clause)

where $S \subseteq \mathbb{N}$ and l, l', l_i are propositional literals with $1 \leq i \leq n$, $n \in \mathbb{N}$. We write $\star : \varphi$ instead of $\mathbb{N} : \varphi$ and such clauses are called *global clauses*. Positive and negative modal clauses are together known as *modal clauses*.

Given a rooted tree Kripke structure M and a set S of natural numbers, by $M[S]$ we denote the set of worlds that are at a modal level in S, that is, $M[S] = \{w \in W \mid \mathsf{ml}_M(w) \in S\}$. Then

$$M \models S : \varphi \text{ iff } \langle M, w\rangle \models \varphi \text{ for every world } w \in M[S].$$

The use of sets as labels allows a concise representation of clauses that might hold in a possibly infinite number of levels.

If $M \models S : \varphi$, then we say that $S : \varphi$ *holds in* M or *is true in* M. For a set Φ of labelled formulae, $M \models \Phi$ iff $M \models S : \varphi$ for every $S : \varphi$ in Φ, and we say Φ is K*-satisfiable*.

We introduce some notation that will be used in the following. For $m, n \in \mathbb{N}$, $m \leq n$, let $[m..n] = \{m, \ldots, n\} \subseteq \mathbb{N}$. Let $S^{+} = \{l + 1 \in \mathbb{N} \mid l \in S\}$, $S^{-} = \{l - 1 \in \mathbb{N} \mid l \in S\}$, and $S^{\geq} = \{ln \in \mathbb{N} \mid n \geq \min(S) \geq l\}$, where $\min(S)$ is the least element in S. Note that the restriction of the elements being in $\mathbb{N}$ implies that S^{-} cannot contain negative numbers.

A formula is in *Separated Normal Form with Modal Levels* (SNF_{ml}) [22,23], if it is a conjunction of clauses in on of the following forms:

$ml : \bigvee_{i=1}^{n} l_i$	$ml : l' \to \Box l$	$ml : l' \to \Diamond l$
(literal clause)	(positive modal clause)	(negative modal clause)

where $ml \in \mathbb{N} \cup \{\star\}$ and l, l', l_i are propositional literals with $1 \leq i \leq n$, $n \in \mathbb{N}$. Effectively, this normal form corresponds to a restriction on the SNF_{sml} where the sets are singletons or $\star$, representing all levels.

3 Reductions

3.1 Definitional Reduction

In [27] we introduced a reduction $\rho_L^{sml}(\varphi)$ that for any modal logic $L = \mathsf{K}\Sigma$ with $\Sigma \subseteq \{\mathsf{B}, \mathsf{D}, \mathsf{T}, \mathsf{4}, \mathsf{5}\}$, transforms a modal formula φ in simplified NNF to a finite set Φ_L^{sml} of clauses in SNF_{sml} such that φ is L-satisfiable iff Φ_L^{sml} is K-satisfiable. For $\mathsf{K4}$, $\mathsf{K5}$ and their extensions by further axioms, ρ_L^{sml} produces sets of clauses where the labelling sets S are potentially infinite. However, depending on syntactic properties of φ it is possible to impose upper bounds on the maximal modal level that occurs in those sets so that the reduction remains satisfiability preserving. Table 3 shows such a bound for each logic in the modal cube. In the table and in the following, for a modal formula φ in simplified NNF, (i) d_m^φ is the modal depth of φ, (ii) $d_\Diamond^\varphi$ is the maximal nesting of $\Diamond$-operators not in the scope of any $\Box$ operators in φ, (iii) $n_\Box^\varphi$ is the number of $\Box$-subformulae in φ, and (iv) $n_\Diamond^\varphi$ is the number of $\Diamond$-subformulae below $\Box$-operators in φ. Using these bounds it is then possible to define a function ρ_L^{ml} that transforms a modal formula φ in simplified NNF to a finite set Φ_L^{ml} of clauses in SNF_{ml} such that φ is L-satisfiable iff Φ_L^{ml} is K-satisfiable.

Table 4 shows the definitions of modified reductions $\bar{\rho}_L^{sml}$ and $\bar{\rho}_L^{ml}$ to SNF_{sml} and SNF_{ml}, respectively. In contrast to ρ_L^{sml}, $\bar{\rho}_L^{sml}$ already uses the bounds in Table 3 to ensure that all labelling sets S occurring in the reduction of a modal formula remain finite. The function $\bar{\rho}_L^{ml}$ then does not enforce further restrictions, but straightforwardly transforms a finite set of SNF_{sml}-clauses with finite labelling sets into a finite set of SNF_{ml} clauses. This presentation of the reduction of modal formulae to a finite set of clauses in SNF_{ml} is closer to the implementation of the process in the prover K$_S$P.

Given a finite set Φ of clauses in SNF_{ml} we can use a function τ^{f} to obtain an equivalent modal formula as follows:

$$\tau^{\mathsf{f}}(\Phi) = \bigwedge\{\Box^{ml} C \mid ml : C \in \Phi\}.$$

where $\Box^0\psi = \psi$ and $\Box^{n+1}\psi = \Box\Box^n\psi$.

Table 3. Bounds on the maximal modal level in SNF_{sml} clauses

Logic L	Bound $d_L^{sml}(\varphi)$
$\mathsf{K}, \mathsf{KD}, \mathsf{KT}, \mathsf{KB}, \mathsf{KDB}, \mathsf{KTB}$	d_m^φ
$\mathsf{K4}, \mathsf{S4}$	$1 + d_\Diamond^\varphi + n_\Diamond^\varphi \times n_\Box^\varphi$
$\mathsf{KD4}$	$1 + d_\Diamond^\varphi + (\max(1, n_\Diamond^\varphi) \times n_\Box^\varphi)$
$\mathsf{KB4}, \mathsf{K5}, \mathsf{S5}, \mathsf{K45}$	$1 + d_\Diamond^\varphi + n_\Diamond^\varphi$
$\mathsf{KD5}, \mathsf{KD45}$	$1 + d_\Diamond^\varphi + \max(1, n_\Diamond^\varphi)$

$$\bar{\rho}_L^{ml}(\varphi) = \{ml : \psi \mid S : \psi \in \bar{\rho}_L^{sml}(\varphi) \text{ and } ml \in S\}$$
$$\bar{\rho}_L^{sml}(\varphi) = \{\{0\} : t_\varphi\} \cup \rho_L^d(\{0\} : t_\varphi \to \varphi)$$

where $d = d_L^{sml}(\varphi)$ as per Table 3 and ρ_L^d is defined as follows:

$$\begin{aligned}
\rho_L^d(S : t \to \mathbf{true}) &= \emptyset \\
\rho_L^d(S : t \to \mathbf{false}) &= \{S : \neg t\} \\
\rho_L^d(S : t \to (\psi_1 \wedge \psi_2)) &= \{S : \neg t \vee \eta(\psi_1), S : \neg t \vee \eta(\psi_2)\} \cup \delta_L^d(S, \psi_1) \cup \delta_L^d(S, \psi_2) \\
\rho_L^d(S : t \to \psi) &= \{S : \neg t \vee \psi\} \\
&\ \text{if } \psi \text{ is a disjunction of literals} \\
\rho_L^d(S : t \to (\psi_1 \vee \psi_2)) &= \{S : \neg t \vee \eta(\psi_1) \vee \eta(\psi_2)\} \cup \delta_L^d(S, \psi_1) \cup \delta_L^d(S, \psi_2) \\
&\ \text{if } \psi_1 \vee \psi_2 \text{ is not a disjunction of literals} \\
\rho_L^d(S : t \to \Diamond\psi) &= \{S : t \to \Diamond\eta(\psi)\} \cup \delta_L^d(S^+, \psi) \\
\rho_L^d(S : t \to \Box\psi) &= P_L^d(S : t \to \Box\psi) \cup \delta_L^d(l\delta_L^d(S), \psi)
\end{aligned}$$

where η and δ_L^d are defined as follows:

$$\eta(\psi) = \begin{cases} \psi, & \text{if } \psi \text{ is a literal} \\ t_\psi, & \text{otherwise} \end{cases} \qquad \delta_L^d(S, \psi) = \begin{cases} \emptyset, & \text{if } \psi \text{ is a literal} \\ \rho_L^d(S : t_\psi \to \psi), & \text{otherwise} \end{cases}$$

and functions P_L^d, lP_L^d and $l\delta_L^d$ are defined as follows:

L	$P_L^d(S : t_{\Box\psi} \to \Box\psi)$	$lP_L^d(S)$	$l\delta_L^d(S)$
K	$S : t_{\Box\psi} \to \Box\eta(\psi)$	S	$S^+ \cap [0..d]$
KB	$S : t_{\Box\psi} \to \Box\eta(\psi)$, $S^- : \eta(\psi) \vee t_{\Box\neg t_{\Box\psi}}$, $S^- : t_{\Box\neg t_{\Box\psi}} \to \Box\neg t_{\Box\psi}$	S	$(S^- \cup S^+)$ $\cap[0..d]$
K4	$S^{\geq} \cap [0..d] : t_{\Box\psi} \to \Box\eta(\psi)$, $S^{\geq} \cap [0..d] : t_{\Box\psi} \to \Box t_{\Box\psi}$	$S^{\geq}$ $\cap[0..d]$	$(S^+)^{\geq}$ $\cap[0..d]$
K5	$[0..d] : t_{\Box\psi} \to \Box\eta(\psi)$, $[0..d] : \neg t_{\Diamond t_{\Box\psi}} \vee t_{\Box\psi}$, $[0..d] : t_{\Diamond t_{\Box\psi}} \to \Diamond t_{\Box\psi}$, $[0..d] : \neg t_{\Diamond t_{\Box\psi}} \to \Box\neg t_{\Box\psi}$, $[0..d] : t_{\Diamond t_{\Box\psi}} \to \Box t_{\Diamond t_{\Box\psi}}$	$[0..d]$	$[0..d]$
KB4	$[0..d] : t_{\Box\psi} \to \Box\eta(\psi)$, $[0..d] : \eta(\psi) \vee t_{\Box\neg t_{\Box\psi}}$, $[0..d] : t_{\Box\psi} \vee t_{\Box\neg t_{\Box\psi}}$, $[0..d] : t_{\Box\neg t_{\Box\psi}} \to \Box\neg t_{\Box\psi}$, $[0..d] : t_{\Box\psi} \to \Box t_{\Box\psi}$	$[0..d]$	$[0..d]$
K45	$[0..d] : t_{\Box\psi} \to \Box\eta(\psi)$, $\{0\} : t_{\Box\psi} \to \Box t_{\Box\psi}$ iff $0 \in S$, $[0..d] : \neg t_{\Diamond t_{\Box\psi}} \vee t_{\Box\psi}$, $[0..d] : t_{\Diamond t_{\Box\psi}} \to \Diamond t_{\Box\psi}$, $[0..d] : \neg t_{\Diamond t_{\Box\psi}} \to \Box\neg t_{\Box\psi}$, $[0..d] : t_{\Diamond t_{\Box\psi}} \to \Box t_{\Diamond t_{\Box\psi}}$	$[0..d]$	$[0..d]$
KDΣ	$\{lP_{\mathsf{K}\Sigma}^d(S) : t_{\Box\psi} \to \Diamond\eta(\psi)\} \cup P_{\mathsf{K}\Sigma}^d(S : t_{\Box\psi} \to \Box\psi)$	–	$l\delta_{\mathsf{K}\Sigma}^d(S)$
KTΣ	$\{lP_{\mathsf{K}\Sigma}^d(S) : \neg t_{\Box\psi} \vee \eta(\psi)\} \cup P_{\mathsf{K}\Sigma}^d(S : t_{\Box\psi} \to \Box\psi)$	–	$l\delta_{\mathsf{K}\Sigma}^d(S) \cup S$

Table 4. $\bar{\rho}_L^{sml}$- and $\bar{\rho}_L^{ml}$-reductions of modal formulae to SNF_{sml} and SNF_{ml}, respectively, $\Sigma \subseteq \{\mathsf{B}, \mathsf{4}, \mathsf{5}\}$.

A smaller equivalent formula can be constructed as follows. For a finite set Φ of clauses in SNF_{ml} let $\Phi[ml] = \{C \mid ml : C \in \Phi\}$ and $ml_{max} = \max\{ml \mid ml : C \in \Phi\}$. Then

$$\tau^{\mathsf{n}}(\Phi) = \bigwedge \Phi[0] \wedge \Box(\bigwedge \Phi[1] \wedge \Box(\bigwedge \Phi[2] \wedge \cdots \wedge \Box(\bigwedge \Phi[ml_{max}]) \cdots)). \qquad (1)$$

Combining $\bar{\rho}_L^{ml}$ and τ^{n} we can define a reduction ρ_L^{def} as

$$\rho_L^{def}(\varphi) = \tau^{\mathsf{n}}(\bar{\rho}_L^{ml}(\varphi))$$

which we call the *definitional reduction* of φ for the modal logic L.

Theorem 1 **([30]).** *Let $L = \mathsf{K}\Sigma$ with $\Sigma \subseteq \{\mathsf{B}, \mathsf{D}, \mathsf{T}, \mathsf{4}, \mathsf{5}\}$ and φ be a modal formula in simplified NNF. Then φ is L-satisfiable iff $\rho_L^{def}(\varphi)$ is K-satisfiable.*

This reduction allows us to use any reasoner for the basic modal logic K as a reasoner for all the logics in the modal cube.

3.2 Axiomatic Reduction

The reductions ρ_L^{sml} and ρ_L^{ml} in [27] were developed as an alternative to and improvement on reductions from the modal logics KB, KD, KT, and $\mathsf{K4}$ to K introduced by Kracht [18]. In contrast to ρ_L^{sml} and ρ_L^{ml} which require modal formulae to be in NNF and treat the modal operators $\Box$ and $\Diamond$ differently, Kracht's reductions assumes that (i) modal formulae are not necessarily in NNF and (ii) the only modal operator occurring in modal formulae is $\Box$ and no distinction is made between positive and negative occurrences of this operator. In the following we extend Kracht's reduction to all logics in the modal cube while adhering to those two assumptions.

Let $\boxdot^{\leq 0}\psi = \psi$ and $\boxdot^{\leq n+1}\psi = (\psi \wedge \Box\boxdot^{\leq n}\psi)$. We can then define a reduction ρ_L^{ax} for all modal logics L in the modal cube as follows:

$$\rho_L^{ax}(\varphi) = \varphi \wedge \boxdot^{\leq b_L^{ax}(\varphi)} \bigwedge P_L^{ax}(\varphi) \qquad (2)$$

Table 5. P_L^{ax}-reduction of $\Box$-formulae, $\Sigma \subseteq \{\mathsf{B}, \mathsf{4}, \mathsf{5}\}$.

L	$P_L^{ax}(\varphi)$	$b_L^{ax}(\varphi)$
K	$\{\mathbf{true}\}$	d_m^{φ}
KB	$\{\neg\psi \rightarrow \Box\neg\Box\psi \mid \Box\psi \in \mathsf{sf}(\varphi)\}$	d_m^{φ}
$\mathsf{K4}$	$\{\Box\psi \rightarrow \Box\Box\psi \mid \Box\psi \in \mathsf{sf}(\varphi)\}$	$n_\Box^{\varphi}$
$\mathsf{K5}$	$\{\neg\Box\neg\Box\psi \rightarrow \Box\psi, \neg\Box\neg\Box\psi \rightarrow \Box\Box\psi, \Box(\Box\psi \rightarrow \Box\Box\psi) \mid \Box\psi \in \mathsf{sf}(\varphi)\}$	1
$\mathsf{KB4}$	$\{\Box\psi \vee \Box\neg\Box\psi \mid \Box\psi \in \mathsf{sf}(\varphi)\} \cup P_{\mathsf{K4}}^{\mathsf{ax}}(\varphi) \cup P_{\mathsf{KB}}^{\mathsf{ax}}(\varphi)$	0
$\mathsf{K45}$	$P_{\mathsf{K4}}^{\mathsf{ax}}(\varphi) \cup P_{\mathsf{K5}}^{\mathsf{ax}}(\varphi)$	0
$\mathsf{KD}\Sigma$	$\{\neg\Box\mathbf{false}\} \cup P_{\Sigma}^{\mathsf{ax}}(\varphi)$	$b_{\mathsf{K}\Sigma}^{ax}(\varphi)$
$\mathsf{KT}\Sigma$	$\{\Box\psi \rightarrow \psi \mid \Box\psi \in \mathsf{sf}(\varphi)\} \cup P_{\Sigma}^{\mathsf{ax}}(\varphi)$	$b_{\mathsf{K}\Sigma}^{ax}(\varphi)$

where $b_L^{ax}(\varphi)$ and $P_L^{ax}(\varphi)$ are as defined in Table 5. We call $\rho_L^{ax}(\varphi)$ the *axiomatic reduction* of φ for the modal logic L.

Theorem 2 *Let $L = \mathsf{K}\Sigma$ with $\Sigma \subseteq \{\mathsf{B}, \mathsf{D}, \mathsf{T}, \mathsf{4}, \mathsf{5}\}$ and φ be a modal formula in simplified NNF. Then φ is L-satisfiable iff $\rho_L^{ax}(\varphi)$ is* K*-satisfiable.*

Just as the definitional reduction, the axiomatic reduction allows us to use any reasoner for basic modal logic as a reasoner for all the logics in the modal cube.

3.3 Discussion

There are five main differences between the definitional reduction and the axiomatic reduction, and between the axiomatic reduction and the work in [18]:

1. The axiomatic reduction for all logics except the logics KB, KD, KT, K4 is new. Kracht [18] did define a reduction from K5 to K4, but since K5 is not a subset of K4, this reduction is not correct. Our definition of the axiomatic reduction corrects that mistake while remaining close to the Kracht's original idea by adding instances of 4 at modal levels greater than 0.
   The bounds given for KB, KD, and KT given in Table 5 are the same as Kracht's [18]. However, for K4 he used a bound given by the number of distinct subformulae of the formula φ under consideration. We are able to show that a bound given by the number of distinct $\Box$-subformulae is sufficient. For the remaining logics, the bounds are new.
2. The definitional reduction introduces new propositional symbols for complex subformulae, so-called *surrogate propositional symbols*. For the modal resolution calculi implemented in K$_S$P [22,26] this is necessary to obtain the clausal normal form on which the calculi operate. However, in the context of our reductions, where we have to add instances of axiom schemata for $\Box$-subformulae, the use of surrogate propositional symbols offers the advantage that repeated occurrences of the same complex subformula can be replaced by the same surrogate symbol. Each surrogate propositional symbol then requires a definition at every modal level at which it occurs, but overall there should still be a benefit in relation to the size of the resulting formula.
3. The bounds shown in Table 3 for the definitional reduction and in Table 5 for the axiomatic reduction, have different effects on the modal formulae produced. For the definitional reduction, the modal depth of $\rho_L^{def}(\varphi)$ is at most $d_L^{sml}(\varphi_L) + 1$, that is, the bound shown for L in Table 3 plus one. In contrast, for the axiomatic reduction, b_L^{ax} in Table 5 only states the modal depth of $\boxdot^{\leq b_L^{ax}} p_a$ where the propositional symbol p_a will then be replaced by a conjunction of instances of axiom schemata for $\Box$-subformulae of φ. For all logics except K and KD, the modal depth of these axiom schemata will be between d_m^{φ} and $d_m^{\varphi} + 2$. Thus, the overall modal depth of $\rho_L^{ax}(\varphi)$ is bound by $b_L^{ax} + d_m^{\varphi} + 2$, not just by the bound shown in Table 5.
   For example, consider the formula $\Box\Box p$ in KB. Then with the axiomatic reduction we obtain the formula

$$\Box\Box p \wedge \boxdot^{\leq 2}((\neg p \rightarrow \Box\neg\Box p) \wedge (\neg\Box p \rightarrow \Box\neg\Box\Box p))$$

which itself is a formula of modal depth 5. With the definitional reduction, we obtain

$$t_{\Box\Box p} \wedge (t_{\Box\Box p} \rightarrow \Box t_{\Box p}) \wedge \Box(t_{\Box p} \rightarrow \Box p) \wedge (p \vee t_{\Box\neg t_{\Box p}}) \wedge (t_{\Box\neg t_{\Box p}} \rightarrow \Box\neg t_{\Box p})$$

which is a formula of modal depth 2.
Taking this into account we can see that for $L = \mathsf{K}\Sigma$ where $\Sigma \subseteq \{\mathsf{B}, \mathsf{D}, \mathsf{T}\}$ we can expect the modal depth of $\rho_L^{def}(\varphi)$ to be less than or equal to that of $\rho_L^{ax}(\varphi)$, while for the remaining logics of the modal cube it depends on the individual formula which reduction will produce a formula of greater modal depth. Nevertheless, for logics such as K4 we expect that the modal depth of $\rho_L^{ax}(\varphi)$ will often be drastically lower than that of $\rho_L^{def}(\varphi)$.

4. The definitional reduction makes a distinction between $\Box$- and $\Diamond$-operators and only introduces additional clauses for $\Box$-subformulae. For logics except KB4, K5 and their extensions, it also carefully tracks at which modal levels additional clauses are required for which occurrences of surrogate symbols that were introduced for $\Box$-subformulae. The 'price' paid for the fact that for these logics additional clauses are not also introduced for $\Diamond$-subformulae is in the higher bounds for the modal levels up to which additional clauses and definitions of surrogate symbols need to be added. The reason is that the presence of axiom instances for negative occurrences of $\Box$-subformulae in the axiomatic reduction for K4, K5 and their extensions allows the 'back-propagation' of $\Box$-subformulae that occur negatively, namely, if $\neg\Box\psi$ is true at a world w at modal level 2 or higher in a tree K-model of $\rho_L^{ax}(\varphi)$, then it is also true at a predecessor world v of w. Provers that do not construct tree Kripke structures, but general Kripke structure, or use caching, can potentially take advantage of this and construct 'shallower' models. On the hand, the outdegree of worlds increases.
5. The definitional reduction for K45, KD45 and KT45 takes account of the fact that instances of 4 are only required to hold at the root world. At all other worlds, instances of 5 are already sufficient to enforce transitivity of the accessibility relation in Kripke structures for these logics. This restriction to the root world is in line with the construction of the definitional reduction ρ_L^{def} in Eq. 1, namely, that we have different sets of clauses associated with each modal level. In contrast, the construction of the axiomatic reduction ρ_L^{ax} in Eq. 2 assumes that we use the same set of axiom instances at every modal level.

We will revisit the effect that Points 2, 3, and 4 have on the size and modal depths of formulae, on the performance of provers, and the models they may produce in the next section.

4 Evaluation

In our evaluation we compare the effect of using the definitional reduction and the axiomatic reduction as input for three provers for K: `CEGARBox` [10], Spartacus [13], and KSP [24,30]. Spartacus and `CEGARBox` were included as they presented best performance in recent evaluations [10,24–26,29,30] when compared with several other provers with built-in support for modal logics: BDDTab [12], FaCT++ [34], InKreSAT [16], SPASS [33], and Leo-III+**E** [8,31].

We have included two more approaches in the comparison: (i) the *global modal resolution* (GMR) calculi [21] that include specific inference rules for each of the logics in the modal cube, implemented in KSP; (ii) *modal layered resolution* (MLR) calculi [22] together with the reductions given in Table 4, again implemented in KSP. The first is an example of 'native' reasoning in the logics concerned, while the inclusion of latter allows us to investigate the effect of 'internalising' the reduction and having inference rules that operate on modal clauses. Both calculi support several refinements of resolution. We report only results for the ordered refinement (cord) as it was the best performing overall.

The two reductions combined with `CEGARBox`, Spartacus, and KSP and the GMR and MLR calculi in KSP give us a total of eight different approaches.

We have used the benchmarks introduced in [27], which comprise[1] (i) 100 unsatisfiable formulae for each of the logics being considered; these are based on 20 formulae each from 5 classes of the LWB benchmark collection [3] modified so that the formulae for logic L are only unsatisfiable in L and its extensions; and also (ii) 100 formulae that are $\mathsf{S5}$-satisfiable, that is, formulae that are satisfiable in all 15 logics; these consist of 20 formulae each from 5 classes of the LWB benchmark collection.

We have supplied all reductions and provers with preprocessed formulae extracted from KSP. The simplified negation normal form for a formula φ, $\mathsf{nnf}(\varphi)$, is generated by KSP as follows. First, the formula is rewritten into box normal form [28], a normal form similar to the negation normal form, but where the operator $\Diamond$ is rewritten as $\neg\Box\neg$. To the resulting formula, we apply prenexing [20], that is, moving the modal operators outwards as much as possible. The simplification rules given in Table 2 are then applied together with pure literal elimination (i.e. replacing occurrences of pure literals by **true**) and constant propagation. Table 6 shows the effect of all these preprocessing steps on average size, average modal depth, and average number of boxes in our benchmark formulae, separately for unsatisfiable (U) and satisfiable (S) formulae. Over all formulae we get a 20% reduction in size and a 66% reduction in the number of $\Box$-operators. The modal depth remains unchanged which is an indication of the robustness of the benchmarks.

For the axiomatic reduction, the resulting formula is then extracted from KSP and the reduction according to Eq. 2 and Table 5 is applied externally. For the

Table 6. Effect of preprocessing on benchmark formulae

Sat	Original Formulae			Simplified Formulae		
	Avg Size	Avg Mod. Depth	Avg #Boxes	Avg Size	Avg Mod. Depth	Avg #Boxes
U	17931	16	405	15549	16	241
S	3641	48	719	1979	48	146

[1] Input files for the provers used here and the source for KSP are available at http://nalon.org/#software.

definitional reduction, the formula is not extracted but transformed by KSP into SNF_{ml} according to Tables 3 and 4. During the transformation into the normal form, complex subformulae are replaced by the same symbol in all positions they might occur. After transformation into SNF_{ml}, the kept clauses are extracted from KSP and used to produce the modal formula for the definitional reduction according to Eq. 1.

Table 7 shows experimental results comparing the performance of the eight approaches. The first three columns of the table show the logic, the satisfiability status of the formulae for our benchmark collection used for this logic ('U' for 'unsatisfiable, 'S' for 'satisfiable'), and their number. In total we have 30 *sets of benchmark formulae*. The next eight columns then show how many of those formulae were solved by each of the eight approaches. A time limit of 100 CPU seconds was set for each formula and where a reduction is used the time taken includes the computation of the reduction. The highest number or numbers in each row are highlighted in bold. The last six columns show the results for ρ_L^{def} and ρ_L^{ax} combined with CEGARBox, Spartacus, and KSP. Here, for each logic L and each satisfiability status we have indicated with italics which reduction resulted in better performance for each of the three provers. In the following we call each such pair a *comparison point*. Benchmarking was performed on a PC with an AMD Ryzen 5 5600X CPU @ 4.60 GHz max and 64 GB main memory using Fedora release 37 as operating system.

For both satisfiable and unsatisfiable benchmark formulae, the combination of the definitional reduction with CEGARBox performs best. Overall, it solves 25% more formulae than the second best approach, the GMR calculi in KSP. CEGARBox with the definitional reduction also outperforms CEGARBox with the axiomatic reduction on both satisfiable and unsatisfiable benchmark formulae. The same is true for the MLR calculus in KSP when combined with one of the two reductions and for Spartacus on satisfiable benchmark formulae when combined with one of the two reductions.

We can see that the internal transformation to SNF_{ml} together with the MLR calculus in KSP performs better than first computing the definitional reduction ρ_L^{def} and then handing the resulting formula to KSP. The former approach performs better on 26 out of 30 sets of benchmark formulae. This is not surprising since in the latter case KSP does apply the transformation into SNF_{ml} again. This implies that new propositional symbols are introduced when applying renaming and new clauses are added defining those symbols. Also, for the ordered resolution refinement we use, all literals in the scope of modal operators will be renamed in order to retain completeness [22]. Again, for each renamed literal there will be an additional clause. Overall, KSP will perform inferences with a larger set of SNF_{ml} clauses over a larger set of propositional symbols. This is bound to degrade performance in most cases.

Looking at individual logics, a more varied picture is evident. Consider both satisfiable and unsatisfiable benchmark formulae for the logics K5, KD5, K4B (which is the same logic as K5B), K45, KD45, and S5 and the behaviour of Spartacus and KSP with one of the two reductions on these. Of these 24 comparison points, the axiomatic reduction results in better performance on 21 and

Table 7. Performance of $\mathsf{K}\Sigma$ provers, ρ_L^{def} combined with K provers and ρ_L^{ax} combined with K provers

L	S	Total	$\mathrm{K_SP}$ (GMR)	$\mathrm{K_SP}$ (MLR)	CEGARBox		Spartacus		$\mathrm{K_SP}$ (MLR)	
					ρ_L^{def}	ρ_L^{ax}	ρ_L^{def}	ρ_L^{ax}	ρ_L^{def}	ρ_L^{ax}
K	S	100	85	**100**	**100**	**100**	**100**	**100**	**100**	**100**
KD	S	100	85	**100**	**100**	62	92	24	**100**	51
KT	S	100	81	68	**100**	63	65	43	65	38
KB	S	100	58	64	**100**	64	**100**	46	65	36
K4	S	100	**85**	58	65	50	56	47	50	18
K5	S	100	60	38	88	**94**	45	73	22	27
KDB	S	100	70	73	**100**	30	82	12	65	30
KTB	S	100	60	57	**100**	49	66	16	56	31
KD4	S	100	**85**	54	57	46	26	19	48	18
KD5	S	100	70	47	**86**	78	18	50	32	27
K45	S	100	53	38	88	**90**	45	83	22	37
KB4	S	100	19	38	**94**	84	63	90	34	59
KD45	S	100	66	47	**86**	84	18	61	32	34
S4	S	100	**76**	44	59	41	36	19	37	15
S5	S	100	57	42	**84**	81	37	65	20	39
All	S	1500	1010	868	**1307**	1016	849	748	748	560
L	S	Total	$\mathrm{K_SP}$ (GMR)	$\mathrm{K_SP}$ (MLR)	CEGARBox		Spartacus		$\mathrm{K_SP}$ (MLR)	
					ρ_L^{def}	ρ_L^{ax}	ρ_L^{def}	ρ_L^{ax}	ρ_L^{def}	ρ_L^{ax}
K	U	100	76	78	**90**	**90**	76	88	82	83
KD	U	100	76	75	**89**	73	73	49	78	35
KT	U	100	78	76	**89**	80	70	44	71	31
KB	U	100	79	52	**82**	66	37	32	49	46
K4	U	100	53	30	**57**	54	30	27	11	26
K5	U	100	46	32	**82**	57	7	60	26	27
KDB	U	100	78	53	**82**	40	38	5	48	23
KTB	U	100	77	50	**84**	43	52	17	47	17
KD4	U	100	**59**	35	51	35	3	4	8	14
KD5	U	100	46	45	**77**	59	5	61	35	10
K45	U	100	40	14	**58**	53	2	49	7	26
KB4	U	100	52	32	**87**	64	56	72	33	39
KD45	U	100	43	29	**57**	53	2	49	15	12
S4	U	100	**68**	23	55	48	32	17	14	9
S5	U	100	44	26	**86**	50	3	58	7	9
All	U	1500	915	650	**1126**	865	486	632	520	407

the definitional reduction only on 3. In particular, Spartacus with the axiomatic reduction consistently shows better performance for these logics than with the definitional reduction. In stark contrast, CEGARBox with the definitional reduction still performs better on 10 out of 12 comparison points. Interestingly, this advantage of the axiomatic reduction does not carry over to K4 and its extensions KD4 and S4. Here, with exceptions of 3 out of 18 comparison points, the definitional reduction with one of CEGARBox, KSP, and Spartacus leads to better performance than the axiomatic reduction.

Table 8. Comparison of axiomatic and definitional reduction combined with Spartacus on satisfiable benchmark formulae.

Logic	Reduction	Solved	Solved by both	Formulae		Models		
				Avg size	Avg modal depth	Avg num of worlds	Avg num of edges	Avg depth
K	ρ^{def}	100	100	5022	48	16	21	7
K	ρ^{ax}	100	100	1722	48	16	21	7
KD	ρ^{def}	92	24	881	4	16	22	4
KD	ρ^{ax}	24	24	13178	9	224	7870	5
KT	ρ^{def}	65	43	6746	8	43	1164	4
KT	ρ^{ax}	43	43	54700	17	272	5967	9
KB	ρ^{def}	100	45	37570	32	2	2	0
KB	ρ^{ax}	46	45	1552250	66	143	1037	1
K4	ρ^{def}	56	34	339748	266	15	48	4
K4	ρ^{ax}	47	34	501328	60	241	5322	9
K5	ρ^{def}	45	44	342049	91	102	817	2
K5	ρ^{ax}	73	44	166858	65	19	158	1
KDB	ρ^{def}	82	12	469	3	9	20	2
KDB	ρ^{ax}	12	12	3637	7	255	3513	4
KTB	ρ^{def}	66	16	805	4	14	32	3
KTB	ρ^{ax}	16	16	4277	8	693	4109	5
KD4	ρ^{def}	26	19	20729	35	267	3073	34
KD4	ρ^{ax}	19	19	18457	17	362	6676	11
KD5	ρ^{def}	18	18	4670	8	246	4112	7
KD5	ρ^{ax}	50	18	3784	7	39	334	3
K45	ρ^{def}	45	44	342095	91	101	810	2
K45	ρ^{ax}	83	44	111403	64	15	86	1
KB4	ρ^{def}	63	62	257823	76	36	155	6
KB4	ρ^{ax}	90	62	65461	50	60	709	5
KD45	ρ^{def}	18	18	4689	8	241	4035	7
KD45	ρ^{ax}	61	18	2383	6	36	188	3
S4	ρ^{def}	36	19	25051	40	145	719	22
S4	ρ^{ax}	19	19	23790	18	300	4723	12
S5	ρ^{def}	37	34	19284	17	205	2968	15
S5	ρ^{ax}	65	34	9541	10	195	1243	7

We can gain additional insight by looking in more detail at the behaviour of provers. While this would be most beneficial for CEGARBox, this tool currently only outputs the satisfiability status of formulae but neither models nor proofs. Instead we turn to Spartacus which can output models for satisfiable formulae. Table 8 shows information on the input formulae that were given to Spartacus, resulting from one of our reductions, and the models that Spartacus produced. The first four columns show the logic, the reduction that was used, how many satisfiable benchmark formulae (out of 100) Spartacus was able to solve, and how many formulae it was able to solve with both reductions. The number in the fourth column is not necessarily the minimum of the two numbers in the

Table 9. Comparison of axiomatic and definitional reduction combined with KSP on unsatisfiable benchmark formulae.

Logic	Reduction	Solved	Solved by both	Formulae		Proof Search		
				Avg size	Avg modal depth	Avg num Inferences	Avg proof size	Avg proof max level
K	ρ^{def}	82	82	1386	17	26912	487	17
K	ρ^{ax}	83	82	933	17	20738	256	17
KD	ρ^{def}	78	35	614	9	792	293	9
KD	ρ^{ax}	35	35	7311	18	295516	164	9
KT	ρ^{def}	71	31	2045	9	9787	191	2
KT	ρ^{ax}	31	31	7145	18	101555	238	9
KB	ρ^{def}	49	46	2510	13	27219	749	7
KB	ρ^{ax}	46	46	13882	27	279984	285	8
K4	ρ^{def}	11	11	31716	133	121620	309	5
K4	ρ^{ax}	26	11	7629	23	52571	160	5
K5	ρ^{def}	26	11	4687	17	139615	259	3
K5	ρ^{ax}	27	11	1552	6	78125	377	4
KDB	ρ^{def}	48	21	1391	8	11045	327	3
KDB	ρ^{ax}	23	21	6660	17	244031	252	7
KTB	ρ^{def}	47	17	1575	7	3212	199	2
KTB	ρ^{ax}	17	17	5875	16	247655	392	7
KD4	ρ^{def}	8	8	82247	231	238114	350	6
KD4	ρ^{ax}	14	8	11707	31	109937	275	6
KD5	ρ^{def}	35	10	8340	17	69268	157	3
KD5	ρ^{ax}	10	10	1926	8	200611	303	4
K45	ρ^{def}	7	7	7138	21	450621	372	4
K45	ρ^{ax}	26	7	1665	7	71733	873	5
KB4	ρ^{def}	33	24	7562	21	164844	226	4
KB4	ρ^{ax}	39	24	2724	11	174535	2189	7
KD45	ρ^{def}	15	5	8023	17	107212	555	6
KD45	ρ^{ax}	12	5	1405	7	88261	768	5
S4	ρ^{def}	14	9	63723	215	122671	252	3
S4	ρ^{ax}	9	9	12102	28	255852	289	4
S5	ρ^{def}	7	4	5731	18	221199	210	4
S5	ρ^{ax}	9	4	1301	7	98434	435	5

third column for a particular logic. The next two columns contain the average size and average modal depth of ρ_L^{def} and ρ_L^{ax} where Spartacus solve both. Finally, the last three columns contain the average number of worlds, number of edges, and depth of the models for these formulae. Spartacus uses blocking, even for the modal logic K, and the models it produces are not trees but general graphs. A fine-grained analysis on the level of individual formulae shows that, with the exception of the logic KB4, it is generally the case that the reduction that produces smaller formulae leads Spartacus to produce smaller models, and thereby also leads to more formulae being solved. Only for KB4 are there more instances where a larger formula resulting from a reduction, namely the definitional reduction, lead to smaller models. However, it is still the case that axiomatic reduction then allows more formulae to be solved for KB4.

For unsatisfiable formulae we consider KSP. Table 9 shows information on the input formulae that were given to KSP, resulting from one of our reductions, and the proof search conducted by KSP. The first six columns correspond to those in Table 8. The final three columns contain the average number of inference steps KSP requires to find a proof, the average size of those proofs, and the average maximal modal level of a clause in those proofs. Again we see that the reduction that produces smaller formulae, with few exceptions, also leads KSP to find proofs in fewer inference steps and allows it to solve more formulae.

5 Conclusions

The axiomatic and the definitional reductions from logics in the modal cube to basic modal logic that we have presented in this paper allow any decision procedure for basic modal logic to be used to solve the satisfiability problem in all 15 logics of the modal cube. This is of particular interest as over the last 25 years, a range of decision procedures for basic modal logic have been implemented and improved [2,6,7,10–13,15,34] but only few implemented decision procedures for all logics of the modal cube exist. Our empirical results also indicate that such reductions are not only a theoretical possibility but are effective and efficient: the combination of the definitional reduction with CEGARBox is currently the best performing approach on our collection of benchmark formulae for the modal cube. There are a number of other contributing factors to the efficiency of the approach that are also beneficial outside the context of reductions. Preprocessing techniques such as simplification and prenexing can reduce the size and, in the context of modal logics, the number of modal operators in a modal formula. The use of surrogate propositional symbols and of a clausal normal form allows to again reduce the size and structural complexity of formulae.

Despite the positive empirical results, we nevertheless hope that more provers that natively support all the logics of the modal cube will be implemented. At the moment our comparison is limited to our own resolution-based prover K$_S$P. Support for modal logics except K in other provers is often limited to KD, KT, and S4. A wider range of provers for all logic in the modal cube would allow us to establish the robustness of our empirical results and possibly enable us to identify strength and weaknesses relative to native provers. It would be beneficial if such support for native reasoning in a logics of the modal cube would also include the provisions of proofs for unsatisfiable formulae and models for satisfiable formulae as well as some abstract measure of the computational effort expended in finding those. This is paramount for our ability to explain the behaviour of prover on our benchmarks.

Finally, our collection of benchmark formulae requires further refinement. Some of the satisfiable formulae in that collection seem to allow rather small models and overall do not appear to be sufficiently challenging across all the logics. We will need to investigate whether this can be remedied simply by moving to higher parameter values for these parameterised classes of formulae or whether completely new classes of formulae are required.

References

1. Areces, C., Gennari, R., Heguiabehere, J., de Rijke, M.: Tree-based heuristic in modal theorem proving. In: Horn, W. (ed.) ECAI 2000, pp. 199–203. IOS Press (2000)
2. Balbiani, P., Demri, S.: Prefixed tableaux systems for modal logics with enriched languages. In: Pollack, M.E. (ed.) IJCAI 1997, pp. 190–195. Morgan Kaufmann (1997)

3. Balsiger, P., Heuerding, A., Schwendimann, S.: A benchmark method for the propositional modal logics K, KT, S4. J. Autom. Reasoning **24**(3), 297–317 (2000). https://doi.org/10.1023/A:1006249507577
4. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge (2002)
5. Demri, S., de Nivelle, H.: Deciding regular grammar logics with converse through first-order logic. J. Logic Lang. Inform. **14**(3), 289–329 (2005)
6. Girlando, M., Straßburger, L.: MOIN: a nested sequent theorem prover for intuitionistic modal logics (system description). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12167, pp. 398–407. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_25
7. Giunchiglia, F., Sebastiani, R.: Building decision procedures for modal logics from propositional decision procedures—the case study of modal K. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS, vol. 1104, pp. 583–597. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61511-3_115
8. Gleißner, T., Steen, A.: LEO-III (2022). https://doi.org/10.5281/zenodo.4435994
9. Gleißner, T., Steen, A., Benzmüller, C.: Theorem provers for every normal modal logic. In: Eiter, T., Sands, D. (eds.) LPAR 2017. EPiC Series in Computing, vol. 46, pp. 14–30. EasyChair (2017). https://doi.org/10.29007/jsb9
10. Goré, R., Kikkert, C.: CEGAR-tableaux: improved modal satisfiability via modal clause-learning and SAT. In: Das, A., Negri, S. (eds.) TABLEAUX 2021. LNCS (LNAI), vol. 12842, pp. 74–91. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86059-2_5
11. Goré, R., Nguyen, L.A.: Clausal tableaux for multimodal logics of belief. Fundam. Inform. **94**(1), 21–40 (2009)
12. Goré, R., Olesen, K., Thomson, J.: Implementing tableau calculi using BDDs: BDDTab system description. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 337–343. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_25
13. Götzmann, D., Kaminski, M., Smolka, G.: Spartacus: a tableau prover for hybrid logic. Electron. Notes Theor. Comput. Sci. **262**, 127–139 (2010)
14. Horrocks, I., Hustadt, U., Sattler, U., Schmidt, R.A.: Computational modal logic. In: Blackburn, P., van Benthem, J., Wolter, F. (eds.) Handbook of Modal Logic, chap. 4, pp. 181–245. Elsevier (2006)
15. Hustadt, U., Schmidt, R.A.: MSPASS: modal reasoning by translation and first-order resolution. In: Dyckhoff, R. (ed.) TABLEAUX 2000. LNCS (LNAI), vol. 1847, pp. 67–71. Springer, Heidelberg (2000). https://doi.org/10.1007/10722086_7
16. Kaminski, M., Tebbi, T.: InKreSAT: modal reasoning via incremental reduction to SAT. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 436–442. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_31
17. Kracht, M.: Reducing modal consequence relations. J. Log. Comput. **11**(6), 879–907 (2001)
18. Kracht, M.: Notes on the space requirements for checking satisfiability in modal logics. In: Balbiani, P., Suzuki, N.Y., Wolter, F., Zakaryaschev, M. (eds.) Advances in Modal Logic 4, pp. 243–264. King's College Publications (2003)
19. Nalon, C.: $\mathrm{K_SP}$ (2022). https://www.nalon.org/#software
20. Nalon, C., Dixon, C.: Anti-prenexing and prenexing for modal logics. In: Fisher, M., van der Hoek, W., Konev, B., Lisitsa, A. (eds.) JELIA 2006. LNCS (LNAI), vol. 4160, pp. 333–345. Springer, Heidelberg (2006). https://doi.org/10.1007/11853886_28

21. Nalon, C., Dixon, C.: Clausal resolution for normal modal logics. J. Algorithms **62**, 117–134 (2007)
22. Nalon, C., Dixon, C., Hustadt, U.: Modal resolution: proofs, layers, and refinements. ACM Trans. Comput. Log. **20**(4), 23:1–23:38 (2019)
23. Nalon, C., Hustadt, U., Dixon, C.: A modal-layered resolution calculus for K. In: De Nivelle, H. (ed.) TABLEAUX 2015. LNCS (LNAI), vol. 9323, pp. 185–200. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24312-2_13
24. Nalon, C., Hustadt, U., Dixon, C.: $\mathrm{K_S}$P: a resolution-based prover for multimodal K. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 406–415. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_28
25. Nalon, C., Hustadt, U., Dixon, C.: $\mathrm{K_S}$P: a resolution-based prover for multimodal K, abridged report. In: Sierra, C. (ed.) IJCAI 2017, pp. 4919–4923. IJCAI/AAAI Press (2017). https://doi.org/10.24963/ijcai.2017/694
26. Nalon, C., Hustadt, U., Dixon, C.: $\mathrm{K_S}$P: Architecture, refinements, strategies and experiments. J. Autom. Reason. **64**(3), 461–484 (2020)
27. Nalon, C., Hustadt, U., Papacchini, F., Dixon, C.: Local reductions for the modal cube. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 486–505. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_29
28. Pan, G., Sattler, U., Vardi, M.Y.: BDD-based decision procedures for the modal logic K. J. Appl. Non-Class. Log. **16**(1–2), 169–208 (2006)
29. Papacchini, F., Nalon, C., Hustadt, U., Dixon, C.: Efficient local reductions to basic modal logic. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 76–92. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_5
30. Papacchini, F., Nalon, C., Hustadt, U., Dixon, C.: Local is best: efficient reductions to modal logic K. J. Autom. Reason. **66**(4), 639–666 (2022). https://doi.org/10.1007/s10817-022-09630-6
31. Schulz, S.: E 2.6 (2022). https://wwwlehre.dhbw-stuttgart.de/~sschulz/E/Download.html
32. Steen, A., Benzmüller, C.: The higher-order prover Leo-III. In: Giacomo, G.D., Catalá, A., Dilkina, B., Milano, M., Barro, S., Bugarín, A., Lang, J. (eds.) ECAI 2020. Frontiers in Artificial Intelligence and Applications, vol. 325, pp. 2937–2938. IOS Press (2020). https://doi.org/10.3233/FAIA200462
33. The SPASS Team: SPASS 3.9 (2016). https://www.spass-prover.org/
34. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: system description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 292–297. Springer, Heidelberg (2006). https://doi.org/10.1007/11814771_26

Left-Linear Completion with AC Axioms

Johannes Niederhauser[1](✉), Nao Hirokawa[2], and Aart Middeldorp[1]

[1] Department of Computer Science, Universität Innsbruck, Innsbruck, Austria
johannes.niederhauser@student.uibk.ac.at, aart.middeldorp@uibk.ac.at
[2] School of Information Science, JAIST, Nomi, Japan
hirokawa@jaist.ac.jp

Abstract. We revisit AC completion for left-linear term rewrite systems where AC unification is avoided and the normal rewrite relation can be used in order to decide validity questions. To that end, we give a new correctness proof for finite runs and establish a simulation result between the two inference systems known from the literature. Furthermore, we show how left-linear AC completion can be simulated by general AC completion. In particular, this result allows us to switch from the former to the latter at any point during a completion process. Finally, we present experimental results for our implementation of left-linear AC completion in the tool accompll.

Keywords: Completion · AC axioms · Term rewriting

1 Introduction

Completion has been extensively studied since its introduction in the seminal paper by Knuth and Bendix [10]. One of the main limitations of the original formulation is its inability to deal with equations which cannot be oriented into a terminating rule such as the commutativity axiom. This shortcoming can be resolved by completion modulo an equational theory $\mathcal{E}$. In the literature, there are two different approaches of achieving this. The general approach [3,6] requires $\mathcal{E}$-unification and allows us to decide validity problems using the rewrite relation $\to_{\mathcal{R}/\mathcal{E}}$ which is defined as $\leftrightarrow_{\mathcal{E}}^{*} \cdot \to_{\mathcal{R}} \cdot \leftrightarrow_{\mathcal{E}}^{*}$. For left-linear term rewrite systems, however, there is Huet's approach [5] which avoids $\mathcal{E}$-unification and allows us to decide validity problems with the normal rewrite relation $\to_{\mathcal{R}}$ and a single check for $\mathcal{E}$-equivalence of the computed normal forms. In their respective books, Avenhaus [1] and Bachmair [3] present inference systems for left-linear completion modulo an equational theory. In this paper, we revisit slightly modified versions (A and B) of these inference systems for finite runs. In addition to a new correctness proof for A in the spirit of [4] which does not rely on proof orderings (Sect. 3), we reduce correctness of B to the correctness of A by establishing a simulation result between finite runs in these systems (Sect. 4). For

Supported by JSPS KAKENHI Grant Number JP22K11900.

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 401–418, 2023.
https://doi.org/10.1007/978-3-031-38499-8_23

the concrete equational theory of associative and commutative (AC) function symbols, we also show the connection between the inference system A and general AC completion by means of another simulation result (Sect. 5). Finally, we present experimental results obtained from our implementation of A for AC in the tool accompll which show that the avoidance of AC unification can result in significant performance improvements over general AC completion (Sects. 6 and 7).

2 Preliminaries

We assume familiarity with term rewriting and completion as described e.g. in [2] but recall some central notions. We consider term rewriting systems (TRSs) which operate on terms over a given signature $\mathcal{F}$. Terms which do not contain the same variable more than once are referred to as *linear* terms. We say that a TRS is *left-linear* if ℓ is a linear term for every rule $\ell \to r \in \mathcal{R}$. A TRS $\mathcal{R}$ is *terminating* if the associated rewrite relation $\to_{\mathcal{R}}$ is well-founded. In that case, we write $s \to^{!}_{\mathcal{R}} t$ if t is a normal form of s. A TRS $\mathcal{R}$ is *confluent* if different computation paths can always be joined, i.e., ${}_{\mathcal{R}}^{*}\!\!\leftarrow \cdot \to^{*}_{\mathcal{R}} \subseteq \to^{*}_{\mathcal{R}} \cdot {}_{\mathcal{R}}^{*}\!\!\leftarrow$. An important sufficient criterion for confluence is the well-known critical pair lemma which states that a terminating TRS is confluent if all non-trivial overlaps between left-hand sides of rules (*critical pairs*) are joinable. Furthermore, there is the notion of *prime critical pairs* [8] which further restricts the considered critical peaks $t \; {}_{\mathcal{R}}^{p}\!\!\leftarrow s \to^{\epsilon}_{\mathcal{R}} u$ to the ones where all proper subterms of $s|_p$ are irreducible. In particular, terminating TRSs whose prime critical pairs are joinable are also confluent. The set of (prime) critical pairs is denoted by $\mathsf{CP}(\mathcal{R})$ ($\mathsf{PCP}(\mathcal{R})$). We define $\mathsf{CP}(\mathcal{R}_1, \mathcal{R}_2)$ as the set of all critical pairs stemming from local peaks of the form $t \; {}_{\mathcal{R}_1}^{p}\!\!\leftarrow s \to^{\epsilon}_{\mathcal{R}_2} u$ and $\mathsf{CP}^{\pm}(\mathcal{R}_1, \mathcal{R}_2) = \mathsf{CP}(\mathcal{R}_1, \mathcal{R}_2) \cup \mathsf{CP}(\mathcal{R}_2, \mathcal{R}_1)$. A TRS is *complete* if it is terminating and confluent. Hence, a complete presentation $\mathcal{R}$ of an equational system (ES) $\mathcal{E}$ can be used to decide the validity problem for $\mathcal{E}$: $s \leftrightarrow^{*}_{\mathcal{E}} t$ if and only if $s \to^{!}_{\mathcal{R}} \cdot {}_{\mathcal{R}}^{!}\!\!\leftarrow t$.

We now turn our attention to rewriting modulo AC function symbols. To that end, we start by giving general definitions for abstract rewrite systems (ARSs). Let $\mathcal{A} = \langle A, \to \rangle$ be an ARS and $\sim$ an equivalence relation on A. We write $\Leftrightarrow$ for $\leftarrow \cup \to \cup \sim$, $\to/\sim$ for $\sim \cdot \to \cdot \sim$ and $\downarrow^{\sim}$ for $\to^{*} \cdot \sim \cdot {}^{*}\!\!\leftarrow$. Given $\mathcal{A}$, we denote $\langle A, \to/\sim \rangle$ by $\mathcal{A}/\sim$. The ARS $\mathcal{A}$ is *terminating modulo* $\sim$ if there are no infinite rewrite sequences with $\to/\sim$ and *Church–Rosser modulo* $\sim$ if $\Leftrightarrow^{*} \subseteq \downarrow^{\sim}$. The ARS $\mathcal{A}$ is *complete modulo* $\sim$ if it is terminating modulo $\sim$ and Church–Rosser modulo $\sim$. While there is no distinction for termination modulo $\sim$ between $\mathcal{A}$ and $\mathcal{A}/\sim$ ($\sim \cdot \sim = \sim$ by transitivity), it makes a considerable difference whether we talk about the Church–Rosser modulo $\sim$ property and therefore completeness modulo $\sim$ of $\mathcal{A}$ or $\mathcal{A}/\sim$. The following lemma is taken from [1, Lemma 4.1.12]. It establishes an important connection between the Church–Rosser modulo $\sim$ property of an ARS $\mathcal{A}$ and $\mathcal{A}/\sim$.

Lemma 1. *Let $\mathcal{A} = \langle A, \rightarrow \rangle$ and $\mathcal{A}' = \langle A, \rightharpoonup \rangle$ be ARSs and $\sim$ an equivalence relation on A such that $\rightarrow \subseteq \rightharpoonup \subseteq \rightarrow/{\sim}$. If $\mathcal{A}'$ is Church–Rosser modulo $\sim$ then $\mathcal{A}/{\sim}$ is Church–Rosser modulo $\sim$.*

The definitions and results for ARSs carry over to TRSs by replacing the equivalence relation $\sim$ by the equational theory $\leftrightarrow_{\mathcal{B}}^{*}$ of an ES $\mathcal{B}$. Most theoretical results of this paper are not specific to AC but hold for an arbitrary base theory $\mathcal{B}$ of which we only demand that $\mathcal{V}\mathsf{ar}(\ell) = \mathcal{V}\mathsf{ar}(r)$ for all $\ell \approx r \in \mathcal{B}$. We abbreviate $\leftrightarrow_{\mathcal{B}}^{*}$ by $\sim_{\mathcal{B}}$ and the rewrite relation $\rightarrow_{\mathcal{R}/\mathcal{B}}$ is defined as $\sim_{\mathcal{B}} \cdot \rightarrow_{\mathcal{R}} \cdot \sim_{\mathcal{B}}$. Furthermore, we write $\downarrow_{\mathcal{R}}^{\sim}$ for the relation $\rightarrow_{\mathcal{R}}^{*} \cdot \sim_{\mathcal{B}} \cdot {}_{\mathcal{R}}^{*}\!\leftarrow$. Termination modulo $\mathcal{B}$ is shown by *$\mathcal{B}$-compatible reduction orders* $>$, i.e., $>$ is well-founded, closed under contexts and substitutions and $\sim_{\mathcal{B}} \cdot > \cdot \sim_{\mathcal{B}} \subseteq >$. This paper deals with a completion procedure which produces TRSs $\mathcal{R}$ such that $\mathcal{R}$ (rather than $\mathcal{R}/\mathcal{B}$) is complete modulo $\mathcal{B}$. In particular, the completion procedure uses the joinability with respect to $\downarrow_{\mathcal{R}}^{\sim}$ of $\mathsf{CP}(\mathcal{R}) \cup \mathsf{CP}^{\pm}(\mathcal{R}, \mathcal{B}^{\pm})$ where $\mathcal{B}^{\pm}$ denotes $\mathcal{B} \cup \{r \approx \ell \mid \ell \approx r \in \mathcal{B}\}$ as a sufficient and necessary criterion for the Church–Rosser modulo $\mathcal{B}$ property of a $\mathcal{B}$-terminating TRS $\mathcal{R}$. Note that this criterion works with standard critical pairs and therefore does not need unification modulo $\mathcal{B}$. However, the criterion is not valid for non-left-linear TRSs as the following example shows.

Example 1. Consider the TRS $\mathcal{R}$ consisting of the single rule $\mathsf{f}(x, x) \rightarrow x$ with $+$ as an additional AC function symbol. There are no critical pairs in $\mathcal{R}$ and between $\mathcal{R}$ and AC, so $\mathsf{CP}(\mathcal{R}) = \mathsf{CP}^{\pm}(\mathcal{R}, \mathsf{AC}^{\pm}) = \varnothing$. Now consider the conversion $\mathsf{f}(x + y, y + x) \sim_{\mathsf{AC}} \mathsf{f}(x + y, x + y) \rightarrow_{\mathcal{R}} x + y$. According to the criterion, $\mathsf{f}(x + y, y + x) \downarrow_{\mathcal{R}}^{\sim} x + y$ should hold, but this is clearly not the case.

3 Avenhaus' Inference System

The idea of completion modulo an equational theory $\mathcal{B}$ for left-linear systems where the normal rewrite relation can be used to decide validity problems has been put forward by Huet [5]. To the best of our knowledge, inference systems for this approach are only presented in the books by Avenhaus [1] and Bachmair [3]. This section presents a new correctness proof of a version of Avenhaus' inference system for finite runs in the spirit of [4] which does not rely on proof orderings. Correctness of Bachmair's system is established by a simulation result in Sect. 4.

3.1 Inference System

Definition 1. *The inference system A is parameterized by a fixed $\mathcal{B}$-compatible reduction order $>$ on terms. It transforms pairs consisting of an ES $\mathcal{E}$ and a TRS $\mathcal{R}$ over the common signature $\mathcal{F}$ according to the following inference rules where $s \dot{\approx} t$ denotes either $s \approx t$ or $t \approx s$:*

$$\textsf{deduce}\quad \frac{\mathcal{E},\mathcal{R}}{\mathcal{E}\cup\{s\approx t\},\mathcal{R}}\ \ \textit{if}\ s\ {}_{\mathcal{R}}\!\leftarrow\cdot\rightarrow_{\mathcal{R}} t \qquad \textsf{orient}\quad \frac{\mathcal{E}\uplus\{s\,\dot{\approx}\,t\},\mathcal{R}}{\mathcal{E},\mathcal{R}\cup\{s\rightarrow t\}}\ \ \textit{if}\ s>t$$

$$\frac{\mathcal{E},\mathcal{R}}{\mathcal{E},\mathcal{R}\cup\{t\rightarrow s\}}\ \ \textit{if}\ s\ {}_{\mathcal{R}}\!\leftarrow\cdot\leftrightarrow_{\mathcal{B}} t \qquad \textsf{delete}\quad \frac{\mathcal{E}\uplus\{s\approx t\},\mathcal{R}}{\mathcal{E},\mathcal{R}}\ \ \textit{if}\ s\sim_{\mathcal{B}} t$$

$$\textsf{simplify}\quad \frac{\mathcal{E}\uplus\{s\,\dot{\approx}\,t\},\mathcal{R}}{\mathcal{E}\cup\{u\approx t\},\mathcal{R}}\ \ \textit{if}\ s\rightarrow_{\mathcal{R}/\mathcal{B}} u \qquad \textsf{collapse}\quad \frac{\mathcal{E},\mathcal{R}\uplus\{t\rightarrow s\}}{\mathcal{E}\cup\{u\approx s\},\mathcal{R}}\ \ \textit{if}\ t\rightarrow_{\mathcal{R}} u$$

$$\textsf{compose}\quad \frac{\mathcal{E},\mathcal{R}\uplus\{s\rightarrow t\}}{\mathcal{E},\mathcal{R}\cup\{s\rightarrow u\}}\ \ \textit{if}\ t\rightarrow_{\mathcal{R}/\mathcal{B}} u$$

A step in an inference system I from an ES $\mathcal{E}$ and a TRS $\mathcal{R}$ to an ES $\mathcal{E}'$ and a TRS $\mathcal{R}'$ is denoted by $(\mathcal{E},\mathcal{R})\vdash_{\mathsf{I}}(\mathcal{E}',\mathcal{R}')$. The parentheses of the pairs are only used when the expression is surrounded by text in order to increase readability. In the following, $\mathsf{PCP}^{\pm}(\mathcal{R},\mathcal{B}^{\pm})$ denotes the restriction of $\mathsf{CP}^{\pm}(\mathcal{R},\mathcal{B}^{\pm})$ to prime critical pairs but where irreducibility is always checked with respect to $\mathcal{R}$, i.e., the critical peaks $t\ {}^{\;p}_{\mathcal{R}}\!\leftarrow s \leftrightarrow^{\epsilon}_{\mathcal{B}} u$ and $t' \leftrightarrow^{p}_{\mathcal{B}} s \rightarrow^{\epsilon}_{\mathcal{R}} u'$ are both prime if all proper subterms of $s|_p$ are irreducible with respect to $\mathcal{R}$.

Definition 2. *Let $\mathcal{E}$ be an ES. A finite sequence*

$$\mathcal{E}_0,\mathcal{R}_0\vdash_{\mathsf{A}}\mathcal{E}_1,\mathcal{R}_1\vdash_{\mathsf{A}}\cdots\vdash_{\mathsf{A}}\mathcal{E}_n,\mathcal{R}_n$$

with $\mathcal{E}_0=\mathcal{E}$ and $\mathcal{R}_0=\varnothing$ is a run *for $\mathcal{E}$. If $\mathcal{E}_n\neq\varnothing$, the run* fails. *The run is* fair *if $\mathcal{R}_n$ is left-linear and the following inclusions hold:*

$$\mathsf{PCP}(\mathcal{R}_n)\subseteq{\downarrow^{\sim}_{\mathcal{R}_n}}\cup\bigcup_{i=0}^{n}\leftrightarrow_{\mathcal{E}_i\cup\mathcal{R}_i}\qquad \mathsf{PCP}^{\pm}(\mathcal{R}_n,\mathcal{B}^{\pm})\subseteq{\downarrow^{\sim}_{\mathcal{R}_n}}\cup\bigcup_{i=0}^{n}\leftrightarrow_{\mathcal{R}_i}$$

Intuitively, fair and non-failing runs yield a $\mathcal{B}$-complete presentation $\mathcal{R}_n$ of the initial set of equations $\mathcal{E}$, i.e., $\leftrightarrow^{*}_{\mathcal{E}\cup\mathcal{B}}=\leftrightarrow^{*}_{\mathcal{R}_n\cup\mathcal{B}}\subseteq{\downarrow^{\sim}_{\mathcal{R}_n}}$. In particular, the inference rules are designed to preserve the equational theory augmented by $\mathcal{B}$. The following example shows that deducing *local cliffs* (${}_{\mathcal{R}}\!\leftarrow\cdot\leftrightarrow_{\mathcal{B}}$) as rules as well as the restriction to $\rightarrow_{\mathcal{R}}$ in the $\mathsf{collapse}$ rule are crucial properties of the inference system.

Example 2. Consider the ES $\mathcal{E}$ consisting of the single equation $x+\mathsf{0}\approx x$ where $+$ is an AC function symbol. We clearly have $\mathsf{0}+x\leftrightarrow^{*}_{\mathcal{E}\cup\mathsf{AC}} x$, so an AC complete system $\mathcal{C}$ representing $\mathcal{E}$ has to satisfy $\mathsf{0}+x\downarrow^{\sim}_{\mathcal{C}} x$. There is just one way to orient the only equation in $\mathcal{E}$, which results in the rule $x+\mathsf{0}\rightarrow x$. Since we want our run to be fair, we add the rules stemming from the prime critical pairs between $x+\mathsf{0}\rightarrow x$ and $\mathsf{AC}^{\pm}$:

$$\mathsf{0}+x\rightarrow x\quad x+(\mathsf{0}+y)\rightarrow x+y\quad x+(y+\mathsf{0})\rightarrow x+y\quad (x+y)+\mathsf{0}\rightarrow x+y$$

If collapsing with $\rightarrow_{\mathcal{R}/\mathsf{AC}}$ is allowed, all these rules become trivial equations and can therefore be deleted. Thus, the modified inference system allows for a fair run which is not complete as $\mathsf{0}+x\downarrow^{\sim}_{\mathcal{R}} x$ does not hold for $\mathcal{R}=\{x+\mathsf{0}\rightarrow x\}$. Furthermore, if we add pairs of terms stemming from local cliffs as equations, we get the same result by applications of $\mathsf{simplify}$.

The inference system presented in Definition 1 is almost the same as the one presented by Avenhaus in [1]. However, since we only consider finite runs, the encompassment condition for the collapse rule has been removed in the spirit of [13]. The following example shows that this can lead to smaller $\mathcal{B}$-complete systems.

Example 3. Consider the ES $\mathcal{E} = \{\mathsf{f}(x + y) \approx \mathsf{f}(x) + \mathsf{f}(y)\}$ where $+$ is an AC symbol. The inference system presented in [1] produces the AC complete system

$$\mathsf{f}(x + y) \to \mathsf{f}(x) + \mathsf{f}(y) \qquad\qquad \mathsf{f}(y + x) \to \mathsf{f}(x) + \mathsf{f}(y)$$

in which either of the rules could be collapsed if it was allowed to collapse with the other rule. In [1] this is prevented by an encompassment condition which essentially forbids to collapse at the root position with a rewrite rule whose left-hand side is a variant of the left-hand side of the rule which should be collapsed. However, this is possible with the system presented in this paper, so for an AC complete representation just one of the two rules suffices.

3.2 Confluence Criterion

The confluence criterion used in the correctness proof of A is an extended version of the one used in [4] which we dub *peak-and-cliff decreasingness.* In the following, we assume that equivalence relations $\sim$ are defined as the reflexive and transitive closure of a symmetric relation $\vdash\!\dashv$, so $\sim\, = \,\vdash\!\dashv^*$. Furthermore, we assume that steps are labeled with labels from a set I, so let $\mathcal{A} = \langle A, \{\to_\alpha\}_{\alpha \in I}\rangle$ be an ARS and $\sim\, = (\bigcup_{\alpha \in I} \vdash\!\dashv_\alpha)^*$an equivalence relation on A.

Definition 3. *The ARS $\mathcal{A}$ is* peak-and-cliff decreasing *if there is a well-founded order $>$ on I such that for all $\alpha, \beta \in I$ the inclusions*

$$_\alpha{\leftarrow} \cdot \to_\beta \;\subseteq\; \xLeftrightarrow[\vee\alpha\beta]{*} \qquad\qquad _\alpha{\leftarrow} \cdot \vdash\!\dashv_\beta \;\subseteq\; \xLeftrightarrow[\vee\alpha]{*} \cdot \xleftarrow[\beta]{=}$$

hold. Here $\vee\alpha\beta$ denotes the set $\{\gamma \in I \mid \alpha > \gamma \,\text{or}\, \beta > \gamma\}$ and if $J \subseteq I$ then $\to_J$ denotes $\bigcup_{\gamma \in J} \to_\gamma$. We simplify $\vee\alpha\alpha$ to $\vee\alpha$.

Lemma 2. *Every conversion modulo $\sim$ is either a valley modulo $\sim$ or contains a local peak or cliff:*

$$\Leftrightarrow^* \;\subseteq\; \downarrow^\sim \cup \Leftrightarrow^* \cdot \leftarrow \cdot \to \cdot \Leftrightarrow^* \cup \Leftrightarrow^* \cdot \vdash\!\dashv \cdot \to \cdot \Leftrightarrow^* \cup \Leftrightarrow^* \cdot \leftarrow \cdot \vdash\!\dashv \cdot \Leftrightarrow^*$$

The proof of the following theorem is based on a well-founded order on multisets. We denote the multiset extension of an order $>$ by $>_{\mathsf{mul}}$. It is well-known that the multiset extension of a well-founded order is also well-founded.

Theorem 1. *If $\mathcal{A}$ is peak-and-cliff decreasing then $\mathcal{A}$ is Church–Rosser modulo $\sim$.*

Proof. With every conversion C we associate a multiset M_C consisting of labels of its rewrite and equivalence relation steps. Since $\mathcal{A}$ is peak-and-cliff decreasing, there is a well-founded order $>$ on I which allows us to replace conversions C of the forms ${}_\alpha\!\leftarrow \cdot \rightarrow_\beta$, ${}_\alpha\!\leftarrow \cdot \vdash\!\dashv_\beta$ and $\vdash\!\dashv_\beta \cdot \rightarrow_\alpha$by conversions C' where $M_C >_{\mathsf{mul}} M_{C'}$. Hence, we prove that $\mathcal{A}$ is Church–Rosser modulo $\sim$, i.e., $\Leftrightarrow^* \subseteq \downarrow^\sim$, by well-founded induction on $>_{\mathsf{mul}}$. Consider a conversion $a \Leftrightarrow^* b$ which we call C. By Lemma 2 we either have $a \downarrow^\sim b$ (which includes the case that C is empty) or one of the following cases holds:

$$a \Leftrightarrow^* \cdot \leftarrow \cdot \rightarrow \cdot \Leftrightarrow^* b \qquad a \Leftrightarrow^* \cdot \leftarrow \cdot \vdash\!\dashv \cdot \Leftrightarrow^* b \qquad a \Leftrightarrow^* \cdot \vdash\!\dashv \cdot \rightarrow \cdot \Leftrightarrow^* b$$

If $a \downarrow^\sim b$ we are immediately done. In the remaining cases, we have a local peak or cliff with concrete labels α and β, so $M_C = \Gamma_1 \uplus \{\alpha, \beta\} \uplus \Gamma_2$. Since $\mathcal{A}$ is peak-and-cliff decreasing, there is a conversion C' with $M_{C'} = \Gamma_1 \uplus \Gamma \uplus \Gamma_3$ where $\{\alpha, \beta\} >_{\mathsf{mul}} \Gamma$. Hence, $M_C >_{\mathsf{mul}} M_{C'}$ and we finish the proof by applying the induction hypothesis. □

In the following, we connect the joinability of local peaks and cliffs to the joinability of prime critical pairs which allows us to apply peak-and-cliff decreasingness in the correctness proof of A.

Definition 4. *Given a TRS $\mathcal{R}$ and terms s, t and u, we write $t \mathrel{\nabla_s} u$ if $s \rightarrow_{\mathcal{R}}^+ t$, $s \rightarrow_{\mathcal{R}}^+ u$, and $t \downarrow_{\mathcal{R}} u$ or $t \leftrightarrow_{\mathsf{PCP}(\mathcal{R})} u$. We write $t \mathrel{\nabla_s^\sim} u$ if $s \rightarrow_{\mathcal{R}}^+ t$, $s \sim u$ and $t \downarrow_{\mathcal{R}}^\sim u$ or $t \leftrightarrow_{\mathsf{PCP}^\pm(\mathcal{R},\mathcal{B}^\pm)} u$. Furthermore, ${}_s^\sim\nabla = \{(u,t) \mid t \mathrel{\nabla_s^\sim} u\}$.*

Lemma 3. *Let $\mathcal{R}$ be a left-linear TRS. The following two properties hold:*

1. *If $t \;{}_{\mathcal{R}}\!\leftarrow s \rightarrow_{\mathcal{R}} u$ then $t \mathrel{\nabla_s^2} u$.*
2. *If $t \;{}_{\mathcal{R}}\!\leftarrow s \leftrightarrow_{\mathcal{B}} u$ then $t \mathrel{\nabla_s} \cdot \mathrel{\nabla_s^\sim} u$.*

3.3 Correctness Proof

We show that every fair and non-failing finite run results in a $\mathcal{B}$-complete presentation. To this end, we first verify that inference steps in A preserve convertibility. We abbreviate $\mathcal{E} \cup \mathcal{R} \cup \mathcal{B}$ to $\mathcal{ERB}$ and $\mathcal{E}' \cup \mathcal{R}' \cup \mathcal{B}$ to $\mathcal{ERB}'$.

Lemma 4. *If $(\mathcal{E}, \mathcal{R}) \vdash_{\mathsf{A}} (\mathcal{E}', \mathcal{R}')$ then the following inclusions hold:*

$$\xrightarrow[\mathcal{ERB}]{} \subseteq \xrightarrow[\mathcal{R}'/\mathcal{B}]{=} \cdot \left(\xrightarrow[\mathcal{ER}']{=} \cup \xleftrightarrow[\mathcal{B}]{*}\right) \cdot \xleftarrow[\mathcal{R}'/\mathcal{B}]{=} \qquad\qquad \xrightarrow[\mathcal{ERB}']{} \subseteq \xleftrightarrow[\mathcal{ERB}]{*}$$

Corollary 1. *If $(\mathcal{E}, \mathcal{R}) \vdash_{\mathsf{A}}^* (\mathcal{E}', \mathcal{R}')$ then $\xleftrightarrow[\mathcal{ERB}]{*} = \xleftrightarrow[\mathcal{ERB}']{*}$.*

Lemma 5. *If $(\mathcal{E}, \mathcal{R}) \vdash_{\mathsf{A}}^* (\mathcal{E}', \mathcal{R}')$ and $\mathcal{R} \subseteq >$ then $\mathcal{R}' \subseteq >$.*

Definition 5. *Let $\leftrightarrow$ be a rewrite relation or equivalence relation, M a finite multiset of terms and $>$ a $\mathcal{B}$-compatible reduction order. We write $s \xleftrightarrow{M} t$ if $s \leftrightarrow t$ and there exist terms $s', t' \in M$ such that $s' \gtrsim s$ and $t' \gtrsim t$ for $\gtrsim \; = \; > \cup \sim_{\mathcal{B}}$.*

We follow the convention that if a conversion is labeled with M, all single steps can be labeled with M.

Lemma 6. *Let $(\mathcal{E},\mathcal{R}) \vdash_{\mathsf{A}} (\mathcal{E}',\mathcal{R}')$ and $\mathcal{R}' \subseteq >$.*

1. *For any finite multiset M we have $\xleftrightarrow[\mathcal{ERB}]{M}{}^{*} \subseteq \xleftrightarrow[\mathcal{ERB}']{M}{}^{*}$.*
2. *If $s \xrightarrow[\mathcal{R}]{M} t$ then $s \xrightarrow[\mathcal{R}']{M}{}^{=} \cdot \xleftrightarrow[\mathcal{ERB}']{N}{}^{*} t$ with $\{s\} >_{\mathsf{mul}} N$.*

Finally, we are able to prove the correctness result for A, i.e., all finite fair and non-failing runs produce a $\mathcal{B}$-complete TRS which represents the original set of equations. In contrast to [1] and [3], the proof shows that it suffices to consider prime critical pairs.

Theorem 2. *Let $\mathcal{E}$ be an ES. For every fair and non-failing run*

$$\mathcal{E}_0,\mathcal{R}_0 \vdash_{\mathsf{A}} \mathcal{E}_1,\mathcal{R}_1 \vdash_{\mathsf{A}} \cdots \vdash_{\mathsf{A}} \mathcal{E}_n,\mathcal{R}_n$$

for $\mathcal{E}$, the TRS $\mathcal{R}_n$ is a $\mathcal{B}$-complete representation of $\mathcal{E}$.

Proof. Let $>$ be the $\mathcal{B}$-compatible reduction order used in the run. From fairness we obtain $\mathcal{E}_n = \varnothing$ as well as the fact that $\mathcal{R}_n$ is left-linear. Corollary 1 establishes $\leftrightarrow^*_{\mathcal{E}\cup\mathcal{B}} = \leftrightarrow^*_{\mathcal{R}_n\cup\mathcal{B}}$ and termination modulo $\mathcal{B}$ of $\mathcal{R}_n$ follows from Lemma 5. It remains to prove that $\mathcal{R}_n$ is Church–Rosser modulo $\mathcal{B}$ which we do by showing peak-and-cliff decreasingness. So consider a labeled local peak $t \;{}_{\mathcal{R}_n}^{M_1}\!\!\leftarrow s \rightarrow_{\mathcal{R}_n}^{M_2} u$. Lemma 3(1) yields $t \triangledown_s^2 u$. Let $v \triangledown_s w$ appear in this sequence (so $v = t$ or $w = u$). By definition, $v \downarrow_{\mathcal{R}_n} w$ or $v \leftrightarrow_{\mathsf{PCP}(\mathcal{R}_n)} w$. Together with fairness, the fact that $\sim_{\mathcal{B}}$ is reflexive as well as closure of rewriting under contexts and substitutions we obtain $v \downarrow^{\sim}_{\mathcal{R}_n} w$ or $(v,w) \in \bigcup_{i=0}^n \leftrightarrow_{\mathcal{E}_i\cup\mathcal{R}_i}$. In both cases, it is possible to label all steps between v and w with $\{v,w\}$. Since $s > v$ and $s > w$ we have $M_1 >_{\mathsf{mul}} \{v,w\}$ and $M_2 >_{\mathsf{mul}} \{v,w\}$. Repeated applications of Lemma 6(1) therefore yield a conversion in $\mathcal{R}_n \cup \mathcal{B}$ between v and w where every step is labeled with a multiset that is smaller than both M_1 and M_2. Hence, the corresponding condition required by peak-and-cliff decreasingness is fulfilled.

Next consider a labeled local cliff $t \;{}_{\mathcal{R}_n}^{M_1}\!\!\leftarrow s \leftrightarrow_{\mathcal{B}}^{M_2} u$. From Lemma 3(2) we obtain a term v such that $t \triangledown_s v \triangledown_s^{\sim} u$. As in the case for local peaks we obtain a conversion between t and v where each step can be labeled with $\{t,v\} <_{\mathsf{mul}} M_1$. Together with fairness, $v \triangledown_s^{\sim} u$ yields $v \downarrow^{\sim}_{\mathcal{R}_n} u$ or $(v,u) \in \bigcup_{i=0}^n \leftrightarrow_{\mathcal{R}_i}$. In the former case there exists a k such that $v \rightarrow^*_{\mathcal{R}_n} \cdot \sim_{\mathcal{B}} \cdot \;{}_{\mathcal{R}_n}^{\;k}\!\!\leftarrow u$. If $k = 0$ we can label all steps with $\{v\}$. If $k > 0$ the conversion is of the form $v \rightarrow^*_{\mathcal{R}_n} \cdot \sim_{\mathcal{B}} \cdot \;{}_{\mathcal{R}_n}^{k-1}\!\!\leftarrow w \;{}_{\mathcal{R}_n}\!\!\leftarrow u$. We can label the rightmost step with M_2 and the remaining steps with $\{v,w\}$. Note that $s > v$. Since $>$ is a $\mathcal{B}$-compatible reduction order we also have $s > w$. Thus, $M_1 >_{\mathsf{mul}} \{v,w\}$ which establishes the corresponding condition required by peak-and-cliff decreasingness for all k. In the remaining case we have $(v,u) \in \bigcup_{i=0}^n \leftrightarrow_{\mathcal{R}_i}$, so there is some $i \leqslant n$ such that $v \leftrightarrow_{\mathcal{R}_i} u$. Actually, we know that $u \rightarrow_{\mathcal{R}_i}^{M_2} v$ since otherwise we would have both $s > v$ and $v > s$ by the $\mathcal{B}$-compatibility of $>$. Repeated applications of Lemma 6(1,2) therefore yield a conversion between u and v of the form

$$u \xrightarrow[\mathcal{R}_n]{M_2}{}^{=} \cdot \xleftrightarrow[\mathcal{R}_n\cup\mathcal{B}]{N}{}^{*} v$$

where $\{u\} >_{\mathsf{mul}} N$. By definition, $s' \gtrsim u$ for some $s' \in M_1$ and therefore $M_1 >_{\mathsf{mul}} N$, which means that the corresponding condition required by peak-and-cliff decreasingness is fulfilled. Overall, it follows that $\mathcal{R}_n$ is peak-and-cliff decreasing and therefore Church–Rosser modulo $\mathcal{B}$. □

Note that the proofs of the previous theorem and Theorem 1 do not require multiset orders induced by quasi-orders but use multiset extensions of proper $\mathcal{B}$-compatible reduction orders which are easier to work with. This could be achieved by defining peak-and-cliff decreasingness in such a way that well-founded orders suffice for the abstract setting. However, the usage of multiset orders based on $\mathcal{B}$-compatible reduction orders as well as a notion of labeled rewriting which allows us to label steps with $\mathcal{B}$-equivalent terms are crucial in order to establish peak-and-cliff decreasingness for TRSs.

4 Bachmair's Inference System

As already mentioned, the inference system proposed by Avenhaus [1] is essentially the same as A. The only other inference system for $\mathcal{B}$-completion for left-linear TRSs is due to Bachmair [3]. We investigate a slightly modified version of this inference system where arbitrary local peaks are deducible and the encompassment condition from the collapse rule is removed as we only consider finite runs and call the resulting system B.

The main difference between A and B is that in B one may only use the standard rewrite relation $\rightarrow_{\mathcal{R}}$ for simplifying equations and composing rules. This allows us to deduce local cliffs as equations. The goal of this section is to establish correctness of B via a simulation by A.

Definition 6. *The inference system* B *is the same as* A *but with rewriting in* compose *and* simplify *restricted to* $\rightarrow_{\mathcal{R}}$ *and the following rule which replaces the two deduction rules of* A*:*

$$\mathsf{deduce}\ \ \frac{\mathcal{E}, \mathcal{R}}{\mathcal{E} \cup \{s \approx t\}, \mathcal{R}} \quad \textit{if } s \ {}_{\mathcal{R}}\!\leftarrow \cdot \rightarrow_{\mathcal{R} \cup \mathcal{B}^{\pm}} t$$

Definition 7. *Let $\mathcal{E}$ be an ES. A finite sequence*

$$\mathcal{E}_0, \mathcal{R}_0 \vdash_{\mathsf{B}} \mathcal{E}_1, \mathcal{R}_1 \vdash_{\mathsf{B}} \cdots \vdash_{\mathsf{B}} \mathcal{E}_n, \mathcal{R}_n$$

with $\mathcal{E}_0 = \mathcal{E}$ and $\mathcal{R}_0 = \varnothing$ is a run *for $\mathcal{E}$. If $\mathcal{E}_n \neq \varnothing$, the run* fails*. The run is* fair *if $\mathcal{R}_n$ is left-linear and the following inclusion holds:*

$$\mathsf{PCP}(\mathcal{R}_n) \cup \mathsf{PCP}^{\pm}(\mathcal{R}_n, \mathcal{B}^{\pm}) \subseteq \downarrow^{\sim}_{\mathcal{R}_n} \cup \bigcup_{i=0}^{n} \leftrightarrow_{\mathcal{E}_i}$$

In contrast to Definition 2, the fairness condition is the same for all prime critical pairs since the inference rule deduce of B never produces rewrite rules.

In the following, $\overset{\mathsf{o}}{\vdash}_{\mathsf{I}}$ denotes an application of the rule orient in an inference system I. In order to prove that fair and non-failing runs in B can be simulated in A, we start with the following technical lemma.

Lemma 7. *If* $(\mathcal{E}_1,\mathcal{R}_1) \vdash_{\mathsf{B}} (\mathcal{E}_2,\mathcal{R}_2)$ *and* $(\mathcal{E}_1,\mathcal{R}_1) \overset{\circ}{\vdash}{}^{*}_{\mathsf{B}} (\mathcal{E}_1',\mathcal{R}_1')$ *then* $(\mathcal{E}_1',\mathcal{R}_1') \vdash^{=}_{\mathsf{A}} (\mathcal{E}_2',\mathcal{R}_2')$ *where* $(\mathcal{E}_2,\mathcal{R}_2) \overset{\circ}{\vdash}{}^{*}_{\mathsf{B}} (\mathcal{E}_2',\mathcal{R}_2')$. *In a picture:*

$$\begin{tikzcd}
\mathcal{E}_1,\mathcal{R}_1 \arrow[r, phantom, "\vdash_{\mathsf{B}}"] \arrow[d, phantom, sloped, "\overset{\circ}{\vdash}{}^{*}_{\mathsf{B}}"] & \mathcal{E}_2,\mathcal{R}_2 \arrow[d, phantom, sloped, "\overset{\circ}{\vdash}{}^{*}_{\mathsf{B}}"] \\
\mathcal{E}_1',\mathcal{R}_1' \arrow[r, phantom, "\vdash^{=}_{\mathsf{A}}"] & \mathcal{E}_2',\mathcal{R}_2'
\end{tikzcd}$$

For the proof of the simulation result, we need a slightly different form of the previous lemma. Analogous to the notation for rewrite relations, the relation $\overset{\circ}{\vdash}{}^{!}$ denotes the exhaustive application of the inference rule orient.

Corollary 2. *If* $(\mathcal{E}_1,\mathcal{R}_1) \vdash_{\mathsf{B}} (\mathcal{E}_2,\mathcal{R}_2)$ *and* $(\mathcal{E}_1,\mathcal{R}_1) \overset{\circ}{\vdash}{}^{!}_{\mathsf{B}} (\mathcal{E}_1',\mathcal{R}_1')$ *then* $(\mathcal{E}_1',\mathcal{R}_1') \vdash^{*}_{\mathsf{A}} (\mathcal{E}_2',\mathcal{R}_2')$ *where* $(\mathcal{E}_2,\mathcal{R}_2) \overset{\circ}{\vdash}{}^{!}_{\mathsf{B}} (\mathcal{E}_2',\mathcal{R}_2')$.

Theorem 3. *For every fair run* $(\mathcal{E},\varnothing) \vdash^{*}_{\mathsf{B}} (\varnothing,\mathcal{R})$ *there exists a fair run* $(\mathcal{E},\varnothing) \vdash^{*}_{\mathsf{A}} (\varnothing,\mathcal{R})$.

Proof. Assume $(\mathcal{E}_0,\mathcal{R}_0) \vdash^{n}_{\mathsf{B}} (\mathcal{E}_n,\mathcal{R}_n)$ where $\mathcal{R}_0 = \mathcal{E}_n = \varnothing$. By n applications of Corollary 2 we arrive at the following situation:

$$\begin{tikzcd}
 & \mathcal{E}_0,\mathcal{R}_0 \arrow[r, phantom, "\vdash_{\mathsf{B}}"] \arrow[d, phantom, sloped, "\overset{\circ}{\vdash}{}^{!}_{\mathsf{B}}"] & \mathcal{E}_1,\mathcal{R}_1 \arrow[r, phantom, "\vdash_{\mathsf{B}}"] \arrow[d, phantom, sloped, "\overset{\circ}{\vdash}{}^{!}_{\mathsf{B}}"] & \cdots \arrow[r, phantom, "\vdash_{\mathsf{B}}"] & \mathcal{E}_n,\mathcal{R}_n \arrow[d, phantom, sloped, "\overset{\circ}{\vdash}{}^{!}_{\mathsf{B}}"] \\
\mathcal{E}_0,\mathcal{R}_0 \arrow[r, phantom, "\overset{\circ}{\vdash}{}^{!}_{\mathsf{A}}"] & \mathcal{E}_0',\mathcal{R}_0' \arrow[r, phantom, "\vdash^{*}_{\mathsf{A}}"] & \mathcal{E}_1',\mathcal{R}_1' \arrow[r, phantom, "\vdash^{*}_{\mathsf{A}}"] & \cdots \arrow[r, phantom, "\vdash^{*}_{\mathsf{A}}"] & \mathcal{E}_n',\mathcal{R}_n'
\end{tikzcd}$$

The following two statements hold:

1. For $0 \leqslant i \leqslant n$, all orientable equations in $\mathcal{E}_i$ are in $\mathcal{R}_i'$ (possibly reversed) and the other equations are in $\mathcal{E}_i'$.
2. $\mathsf{PCP}^{\pm}(\mathcal{R}_n',\mathcal{B}^{\pm})$ is a set of orientable equations.

Statement (1) is immediate from the simulation relation $\overset{\circ}{\vdash}{}^{!}_{\mathsf{B}}$ and statement (2) follows from $\mathcal{B}$-compatibility of the used reduction order together with the fact that every (prime) critical pair is connected by one $\mathcal{R}_n$-step and one $\mathcal{B}$-step. Furthermore, $\mathcal{E}_n = \varnothing$ implies $\mathcal{E}_n' = \varnothing$ as well as $\mathcal{R}_n = \mathcal{R}_n'$. Hence, we obtain fairness of the run in A by showing the following inclusions:

$$\mathsf{PCP}(\mathcal{R}_n') \subseteq {\downarrow^{\sim}_{\mathcal{R}_n'}} \cup \bigcup_{i=0}^{n} \leftrightarrow_{\mathcal{E}_i' \cup \mathcal{R}_i'} \qquad \mathsf{PCP}^{\pm}(\mathcal{R}_n',\mathcal{B}^{\pm}) \subseteq {\downarrow^{\sim}_{\mathcal{R}_n'}} \cup \bigcup_{i=0}^{n} \leftrightarrow_{\mathcal{R}_i'}$$

Let $s \approx t \in \mathsf{PCP}(\mathcal{R}_n')$. By fairness of the run in B we obtain $s \downarrow^{\sim}_{\mathcal{R}_n'} t$ or $s \leftrightarrow_{\mathcal{E}_k} t$ for some $k \leqslant n$. In the former case, we are immediately done. In the latter case we obtain $s \leftrightarrow_{\mathcal{E}'_k \cup \mathcal{R}'_k} t$ from (1) as desired. Now, let $s \approx t \in \mathsf{PCP}^{\pm}(\mathcal{R}'_n, \mathcal{B}^{\pm})$. By fairness of the run in B we obtain $s \downarrow^{\sim}_{\mathcal{R}'_n} t$ or $s \leftrightarrow_{\mathcal{E}_k} t$ for some $k \leqslant n$. Again, we are immediately done in the former case. In the latter case we have $s \leftrightarrow_{\mathcal{R}'_k} t$ because of (1) and (2). Therefore, the run in A is fair. □

The previous theorem is an important simulation result which justifies the emphasis on A in this paper. Moreover, together with Theorem 2 the correctness of the inference system B is an easy consequence.

Corollary 3. *Every fair and non-failing run for $\mathcal{E}$ in* B *produces a $\mathcal{B}$-complete presentation of $\mathcal{E}$.*

5 AC Completion

So far, the theoretical results have been generalized by using the equational theory $\mathcal{B}$ as a placeholder. In practice, however, this paper is concerned with the particular theory AC. The results of this section allow us to assess the effectiveness of the inference system A in the setting of AC completion.

5.1 Limitations of Left-Linear AC Completion

In addition to the restriction to left-linear rewrite rules, the following example demonstrates another severe limitation of the inference system A previously unmentioned in the literature.

Example 4. Consider the ES $\mathcal{E}$ consisting of the equations

$$\mathsf{and}(0,0) \approx 0 \qquad \mathsf{and}(1,1) \approx 1 \qquad \mathsf{and}(0,1) \approx 0$$

where and is an AC function symbol. There is only one way to orient each equation. Furthermore, there are no critical pairs between the resulting rewrite rules. Hence, using the inference system A we arrive at the intermediate TRS

$$\mathsf{and}(0,0) \to 0 \qquad \mathsf{and}(1,1) \to 1 \qquad \mathsf{and}(0,1) \to 0$$

where the only possible next step is to deduce local cliffs. We will now show that this has to be done infinitely many times. Note that an AC-complete presentation $\mathcal{R}$ of $\mathcal{E}$ has to be able to rewrite any AC-equivalent term of a redex: Consider the infinite family of terms

$$s_0 = \mathsf{and}(0,1) \; s_1 = \mathsf{and}(\mathsf{and}(0,x_1),1) \; s_2 = \mathsf{and}(\mathsf{and}(\mathsf{and}(0,x_1),x_2),1) \cdots$$

as well as

$$t_0 = 0 \qquad t_1 = \mathsf{and}(0,x_1) \qquad t_2 = \mathsf{and}(\mathsf{and}(0,x_1),x_2) \qquad \cdots$$

Clearly, $s_n \leftrightarrow^*_{\mathcal{E} \cup \mathsf{AC}} t_n$ for all $n \in \mathbb{N}$ and therefore also $s_n \downarrow^{\sim}_{\mathcal{R}} t_n$ for all $n \in \mathbb{N}$, but this demands infinitely many rules in $\mathcal{R}$: For each s_n there is an AC-equivalent term such that the constants $\mathsf{0}$ and $\mathsf{1}$ are next to each other which allows us to rewrite it using the rule $\mathsf{and}(\mathsf{0},\mathsf{1}) \to \mathsf{0}$. However, with n also the amount of variables between these constants increases which requires $\mathcal{R}$ to have infinitely many rules since rewrite rules can only be applied before the representation modulo AC is changed.

Note that there is nothing special about this example except the fact that it contains at least one equation which can only be oriented such that the left-hand side contains an AC function symbol where both arguments have "structure", i.e., both arguments represent more complicated terms than a variable. As a consequence, the necessity of infinite rules applies to all equational systems which have this property. Needless to say, this means that for a large class of equational systems the corresponding AC-canonical presentation (in the left-linear sense) is infinite if it exists. This observation is in stark contrast to the properties of general AC completion as presented in the next section which can complete the ES $\mathcal{E}$ from Example 4 into a finite AC-canonical TRS by simply orienting all rules from left to right.

5.2 General AC Completion

Inference systems for completion modulo an equational theory which are not restricted to the left-linear case usually need more inference rules than the ones already covered in this paper. For general AC completion, however, there exists a particularly simple inference system which constitutes a special case of normalized completion [12] and can be found in Sarah Winkler's PhD thesis [16, p. 109].

Definition 8. *The inference system* $\mathsf{KB}_{\mathsf{AC}}$ *is the same as* A *for the fixed theory AC but with a modified collapse rule which allows us to rewrite with* $\to_{\mathcal{R}/\mathsf{AC}}$ *and the following rule which replaces the two deduction rules of* A*:*

$$\mathsf{deduce}\quad \frac{\mathcal{E}, \mathcal{R}}{\mathcal{E} \cup \{s \approx t\}, \mathcal{R}} \quad \textit{if } s\ {}_{\mathcal{R}}{\leftarrow} \cdot \sim_{\mathsf{AC}} \cdot \to_{\mathcal{R}} t$$

The purpose of this section is to show how A can be simulated by $\mathsf{KB}_{\mathsf{AC}}$ in the case of $\mathcal{B} = \mathsf{AC}$. Since local cliffs cannot be deduced in $\mathsf{KB}_{\mathsf{AC}}$, the simulation has to work with a potentially smaller set of rewrite rules. Furthermore, during a run, the variants of rules stemming from local cliffs may be in different states with respect to inter-reduction ($\mathsf{collapse}$ and $\mathsf{compose}$). Given an intermediate TRS $\mathcal{R}$ of a run in A as well as an intermediate TRS $\mathcal{R}'$ of a run in $\mathsf{KB}_{\mathsf{AC}}$, the invariant $\mathcal{R} \subseteq \to^{+}_{\mathcal{R}'/\mathsf{AC}}$ resolves both of the aforementioned problems. The main motivation behind this invariant is the avoidance of $\mathsf{compose}$ and $\mathsf{collapse}$ in the $\mathsf{KB}_{\mathsf{AC}}$ run.

Lemma 8. *If* $(\mathcal{E}_1, \mathcal{R}_1) \vdash_{\mathsf{A}} (\mathcal{E}_2, \mathcal{R}_2)$ *and* $\mathcal{R}_1 \subseteq \to^{+}_{\mathcal{R}'_1/\mathsf{AC}}$ *then there exists a TRS* $\mathcal{R}'_2$ *such that* $(\mathcal{E}_1, \mathcal{R}'_1) \vdash^{*}_{\mathsf{KB}_{\mathsf{AC}}} (\mathcal{E}_2, \mathcal{R}'_2)$ *and* $\mathcal{R}_2 \subseteq \to^{+}_{\mathcal{R}'_2/\mathsf{AC}}$.

Proof. Let $>$ be a fixed AC-compatible reduction order which is used in both A and $\mathsf{KB}_{\mathsf{AC}}$. Suppose $(\mathcal{E}_1, \mathcal{R}_1) \vdash_{\mathsf{A}} (\mathcal{E}_2, \mathcal{R}_2)$ and $\mathcal{R}_1 \subseteq \to^{+}_{\mathcal{R}'_1/\mathsf{AC}}$. We proceed by a case analysis on the rule applied in the inference step $(\mathcal{E}_1, \mathcal{R}_1) \vdash_{\mathsf{A}} (\mathcal{E}_2, \mathcal{R}_2)$. The only interesting cases are when deduce, simplify, compose, or collapse is applied.

- If deduce is applied, we further distinguish whether it was applied to a local peak or cliff. In the case of a local cliff, we have $\mathcal{E}_1 = \mathcal{E}_2$ and $\mathcal{R}_2 = \mathcal{R}_1 \cup \{\ell \to r\}$ with $\ell \to_{\mathcal{R}_1/\mathsf{AC}} r$. From $\ell \to_{\mathcal{R}_1/\mathsf{AC}} r$ and $\mathcal{R}_1 \subseteq \to^{+}_{\mathcal{R}'_1/\mathsf{AC}}$ we obtain $\ell \to^{+}_{\mathcal{R}'_1/\mathsf{AC}} r$. Thus, $\mathcal{R}_2 \subseteq \to^{+}_{\mathcal{R}'_1/\mathsf{AC}}$ holds. As $(\mathcal{E}_1, \mathcal{R}'_1) \vdash^{0}_{\mathsf{KB}_{\mathsf{AC}}} (\mathcal{E}_2, \mathcal{R}'_1)$ is trivial, the claim follows. In the case of a local peak, we have $\mathcal{R}_1 = \mathcal{R}_2$ and $\mathcal{E}_2 = \mathcal{E}_1 \cup \{t \approx u\}$ with $t \;{}_{\mathcal{R}_1}{\leftarrow}\; s \to_{\mathcal{R}_1} u$. Since $\mathcal{R}_1 \subseteq \to^{+}_{\mathcal{R}'_1/\mathsf{AC}}$ holds, we have $t \;{}_{\mathcal{R}'_1/\mathsf{AC}}{\overset{*}{\leftarrow}}\; v \;{}_{\mathcal{R}'_1}{\leftarrow}\; \cdot \sim_{\mathsf{AC}} s \sim_{\mathsf{AC}} \cdot \to_{\mathcal{R}'_1} w \to^{*}_{\mathcal{R}'_1/\mathsf{AC}} u$ for some v and w. By performing deduce and simplify steps
$$(\mathcal{E}_1, \mathcal{R}'_1) \vdash_{\mathsf{KB}_{\mathsf{AC}}} (\mathcal{E}_1 \cup \{v \approx w\}, \mathcal{R}'_1) \vdash^{*}_{\mathsf{KB}_{\mathsf{AC}}} (\mathcal{E}_1 \cup \{t \approx u\}, \mathcal{R}'_1) = (\mathcal{E}_2, \mathcal{R}'_1)$$
is obtained. As $\mathcal{R}_1 = \mathcal{R}_2$, the inclusion $\mathcal{R}_2 \subseteq \to^{+}_{\mathcal{R}'_1/\mathsf{AC}}$ is trivial. Hence, the claim holds.
- If simplify is applied, we have $\mathcal{R}_1 = \mathcal{R}_2$, $\mathcal{E}_1 = \mathcal{E}_0 \cup \{s \approx t\}$, and $\mathcal{E}_2 = \mathcal{E}_0 \cup \{s' \approx t'\}$ with $s \to^{=}_{\mathcal{R}_1} s'$ and $t \to^{=}_{\mathcal{R}_1} t'$. By $\mathcal{R}_1 \subseteq \to^{+}_{\mathcal{R}'_1/\mathsf{AC}}$ we have $s \to^{*}_{\mathcal{R}'_1/\mathsf{AC}} s'$ and $t \to^{*}_{\mathcal{R}'_1/\mathsf{AC}} t'$. Therefore, performing simplify, we obtain $(\mathcal{E}_1, \mathcal{R}'_1) \vdash^{*}_{\mathsf{KB}_{\mathsf{AC}}} (\mathcal{E}_2, \mathcal{R}'_1)$. As $\mathcal{R}_1 = \mathcal{R}_2$, the inclusion $\mathcal{R}_2 \subseteq \to^{+}_{\mathcal{R}'_1/\mathsf{AC}}$ is trivial.
- If compose is applied, we can write $\mathcal{E}_1 = \mathcal{E}_2$, $\mathcal{R}_1 = \mathcal{R}_0 \cup \{\ell \to r\}$, and $\mathcal{R}_2 = \mathcal{R}_0 \cup \{\ell \to r'\}$ with $r \to_{\mathcal{R}_0/\mathsf{AC}} r'$. We have $(\mathcal{E}_1, \mathcal{R}'_1) \vdash^{0}_{\mathsf{KB}_{\mathsf{AC}}} (\mathcal{E}_2, \mathcal{R}'_1)$. Since the inclusions $\mathcal{R}_0 \subseteq \mathcal{R}_1 \subseteq \to^{+}_{\mathcal{R}'_1/\mathsf{AC}}$ yield $\ell \to^{+}_{\mathcal{R}'_1/\mathsf{AC}} r \to^{+}_{\mathcal{R}'_1/\mathsf{AC}} r'$, we obtain $\mathcal{R}_2 \subseteq \to^{+}_{\mathcal{R}'_1/\mathsf{AC}}$.
- If collapse is applied, we can write $\mathcal{E}_2 = \mathcal{E}_1 \cup \{\ell' \approx r\}$ and $\mathcal{R}_1 = \mathcal{R}_2 \uplus \{\ell \to r\}$ with $\ell \to_{\mathcal{R}_2} \ell'$. By $\mathcal{R}_2 \subseteq \mathcal{R}_1 \subseteq \to^{+}_{\mathcal{R}'_1/\mathsf{AC}}$ we have
$$\ell' \;{}_{\mathcal{R}'_1/\mathsf{AC}}{\overset{*}{\leftarrow}}\; t \;{}_{\mathcal{R}'_1}{\leftarrow}\; \cdot \sim_{\mathsf{AC}} \ell \sim_{\mathsf{AC}} \cdot \to_{\mathcal{R}'_1} u \to^{*}_{\mathcal{R}'_1/\mathsf{AC}} r$$
for some t and u. Performing deduce and simplify, we obtain:
$$(\mathcal{E}_1, \mathcal{R}'_1) \vdash_{\mathsf{KB}_{\mathsf{AC}}} (\mathcal{E}_1 \cup \{t \approx u\}, \mathcal{R}'_1) \vdash^{*}_{\mathsf{KB}_{\mathsf{AC}}} (\mathcal{E}_1 \cup \{\ell' \approx r\}, \mathcal{R}'_1) = (\mathcal{E}_2, \mathcal{R}'_1)$$
By $\mathcal{R}_2 \subseteq \mathcal{R}_1 \subseteq \to^{+}_{\mathcal{R}'_1/\mathsf{AC}}$ the claim is concluded. □

Theorem 4. *For every fair run $(\mathcal{E}, \varnothing) \vdash^{*}_{\mathsf{A}} (\varnothing, \mathcal{R})$ there exists a run $(\mathcal{E}, \varnothing) \vdash^{*}_{\mathsf{KB}_{\mathsf{AC}}} (\varnothing, \mathcal{R}')$ such that $\mathcal{R}'/\mathsf{AC}$ is an AC-complete presentation of $\mathcal{E}$.*

Proof. With a straightforward induction argument, we obtain the run $(\mathcal{E}, \varnothing) \vdash^{*}_{\mathsf{KB}_{\mathsf{AC}}} (\varnothing, \mathcal{R}')$ as well as $\mathcal{R} \subseteq \to^{+}_{\mathcal{R}'/\mathsf{AC}}$ $(*)$ from Lemma 8. Furthermore, AC termination of $\mathcal{R}'$ and $\leftrightarrow^{*}_{\mathcal{E} \cup \mathsf{AC}} = \leftrightarrow^{*}_{\mathcal{R}' \cup \mathsf{AC}}$ $(**)$ are easy consequences from the definition of $\mathsf{KB}_{\mathsf{AC}}$. AC-completeness of $\mathcal{R}$ follows from fairness of the run in A and Theorem 2. For the Church–Rosser modulo AC property of $\mathcal{R}'/\mathsf{AC}$, consider a conversion $s \leftrightarrow^*_{\mathcal{R}' \cup \mathsf{AC}} t$. From $(**)$ we obtain $s \leftrightarrow^*_{\mathcal{E} \cup \mathsf{AC}}$ and therefore $s \to^*_{\mathcal{R}} \cdot \sim_{\mathsf{AC}} \cdot {}_{\mathcal{R}}^{*}\!\!\leftarrow t$ by the fact that $\mathcal{R}$ is an AC-complete presentation of $\mathcal{E}$. Finally, $(*)$ yields $s \to^*_{\mathcal{R}'/\mathsf{AC}} \cdot \sim_{\mathsf{AC}} \cdot {}_{\mathcal{R}'/\mathsf{AC}}^{*}\!\!\leftarrow t$ as desired. Thus, $\mathcal{R}'/\mathsf{AC}$ is an AC-complete presentation of $\mathcal{E}$. □

In addition to the result of the previous theorem, the proof of Lemma 8 provides a procedure to construct a $\mathsf{KB}_{\mathsf{AC}}$ run which "corresponds" to a given A run. In particular, this means that it is possible to switch from A to $\mathsf{KB}_{\mathsf{AC}}$ at any point while performing AC completion. This is of practical relevance: Assume that AC completion is started with A in order to avoid AC unification. If A gets stuck due to simplified equations which are not orientable into a left-linear rule or it seems to be the case that the procedure diverges due to the problem described in Example 4, starting from scratch with $\mathsf{KB}_{\mathsf{AC}}$ is not necessary. We conclude the section by illustrating the practical relevance of the simulation result with an example.

Example 5. Consider the ES $\mathcal{E}$ for abelian groups consisting of the equations

$$\mathsf{e} \cdot x \approx x \qquad\qquad x^- \cdot x \approx \mathsf{e}$$

where $\cdot$ is an AC symbol. Note that the well-known completion run for non-abelian group theory is also a run in A: Critical pairs with respect to the associativity axiom are deducible via local cliffs, non-left-linear intermediate rules are allowed and all (intermediate) rules are orientable with e.g. AC-KBO. Hence, we obtain the TRS $\mathcal{R}'$ consisting of the rules

$$\begin{array}{rrclcrrcl}
1: & \mathsf{e} \cdot x & \to & x & \qquad & 6: & x \cdot \mathsf{e} & \to & x \\
2: & x^- \cdot x & \to & \mathsf{e} & & 7: & x \cdot x^- & \to & \mathsf{e} \\
3: & x^{--} & \to & x & & 8: & \mathsf{e}^- & \to & \mathsf{e} \\
4: & x^- \cdot (x \cdot y) & \to & y & & 9: & x \cdot (x^- \cdot y) & \to & y \\
5: & (x \cdot y)^- & \to & y^- \cdot x^- & & & & &
\end{array}$$

and switch to $\mathsf{KB}_{\mathsf{AC}}$ where we can collapse the redundant rules 4, 6, 7 and 9. A final joinability check of all AC critical pairs reveals that the resulting TRS $\mathcal{R}$ is an AC-complete presentation of abelian groups. Hence, the simulation result allows to make progress with A even when it is doomed to fail. In particular, critical pairs between rules whose left-hand sides do not contain AC symbols do not need to be recomputed.

6 Implementation

To the best of our knowledge, our tool $\mathsf{accompll}$ is the first implementation of left-linear AC completion. It is written in Haskell and available on its website[1]. Instead of expecting explicit AC-compatible reduction orders as input, accompll performs completion with termination tools [15]. In principle, completion with termination tools has to consider all combinations of possible orientations of equations in order to find a complete system. However, traversing the whole search space is rather inefficient. The state of the art for solving this problem efficiently is *multi-completion with termination tools* due to Winkler et al. [20]. Since the implementation of this method is a major effort, accompll adopts a simple but incomplete strategy presented in [14]: Instead of traversing the whole search space, accompll runs two threads in parallel where one thread prefers to orient equations from left to right and vice versa. If one of the threads finishes successfully, the corresponding result is reported. Completion fails if both threads fail.

As input, the tool expects a file in the WST[2] format describing the equational theory on which left-linear AC completion should be performed. The user can choose whether $\to_{\mathcal{R}}$ or $\to_{\mathcal{R}/\mathsf{AC}}$ is used for rewriting in the inference rules simplify and compose. Furthermore, the generation of critical pairs can be restricted to the primality criterion.

Another feature is the validity problem solving mode which solves a given instance of the validity problem for an equational theory $\mathcal{E}$ upon successful completion of $\mathcal{E}$. This mode can be triggered by supplying a concrete equation $s \approx t$ as a command line argument in addition to the file describing $\mathcal{E}$.

In the tool accompll, external termination tools do much of the heavy lifting. In particular, the user can supply the executable of an arbitrary termination tool as long as the output starts with `YES`, `MAYBE`, `NO` or `TIMEOUT` (all other cases are treated as an error). The input format for the termination tool can be set by a command line argument. The available options are the WST format as well as the XML format of the Nagoya Termination Tool [21][3].

Since starting a new process for every call of the termination tool causes a lot of operating system overhead, the tool supports an interactive mode which allows it to communicate with a single process of the termination tool in a dialogue style. Here, the only constraint for the termination tool is that it accepts a sequence of termination problems separated by the keyword `(RUN)`. This is currently only implemented in an experimental version of Tyrolean Termination Tool 2 (TTT2) [11], but we hope that more termination tools will follow as this approach has a positive effect on the runtime of completion with termination tools while demanding comparatively little implementation effort.

7 Experimental Results

The problem set used for the experimental results consists of 50 ESs. It is based on the one used in [18] and has been extended by further examples from the literature as well as handcrafted examples. The experiments were performed on

[1] https://github.com/niedjoh/accompll.

[2] https://www.lri.fr/~marche/tpdb/format.html.

[3] https://www.trs.cm.is.nagoya-u.ac.jp/NaTT/natt-xml.html.

Table 1. Experimental results on 50 problems (excerpt)

	accompll ($\mathsf{T_TT_2}$)		accompll ($\mathsf{T_TT_2}$e)		MædMax		mkbTT	
	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)
ℕ (+, ×)	0.85	10	0.28	10	18.78	5	∞	
ℕ (+, −, ×, ÷)	1.74	15	0.42	15	∞		60.06	?[a]
[1, Ex. 4.2.15(b)]	0.48	4	0.24	4	0.01	3	0.19	2
abelian groups	⊥		⊥		0.16	5	0.14	0
[7, Ex. 2]	∞		∞		0.04	5	0.44	3
problems solved	**16**		**16**		**22**		**35**	

[a] mkbTT does not output the completed system for unknown reasons.

an Intel Core i7-7500U running at a clock rate of 2.7 GHz with 15.5 GiB of main memory. Our tool accompll was used with the termination tool $\mathsf{T_TT_2}$ as well as an experimental version (denoted by $\mathsf{T_TT_2}$e) which allows our tool to communicate a sequence of termination problems without having to start a new process all the time, as described in the preceding section.

Table 1 shows some interesting results and compares the two configurations of accompll with the normalized completion [12] mode of mkbTT [19] and the AC completion mode of MædMax [17]. The tool mkbTT is the original implementation of multi-completion with termination tools [20]. MædMax, on the other hand, implements *maximal completion* [9] which makes use of MaxSAT/MaxSMT solvers instead of termination tools in order to avoid using concrete reduction orders as input. To the best of our knowledge, there is no comparable completion tool which supports AC axioms. Since normalized completion subsumes general AC completion, a comparison with the aforementioned modes of both systems allows us to assess the effectiveness of accompll with respect to the state of the art in AC completion. Note that normalized completion uses AC unification.

In Table 1, columns (1) show the execution time in seconds where ∞ denotes that the timeout of 60 s has been reached and ⊥ denotes failure of completion. Columns (2) state the number of rules of the completed TRS. The first two problems show that the avoidance of AC unification can indeed have a positive effect on the execution time. However, the third problem indicates that there may also be an opposite effect on small problems. The last two problems show the two main limitations of left-linear AC completion: Abelian groups do not have an AC-complete presentation which is left-linear and Example 2 from [7] is a ground ES which causes left-linear AC completion to suffer from the problem described in Example 4 by definition. The severity of these limitations is reflected in the total number of solved problems. In particular, the problem set does not contain an ES which is completed only by accompll. However, given Theorem 4, this is not unexpected. Another noteworthy but unsurprising fact is that complete systems produced by accompll tend to have more rules since every rule needs different versions of left-hand sides to facilitate rewriting without AC-matching. It would also be interesting to compare the execution times for typical queries of the form $\mathcal{E} \vDash s \approx t$ as the resulting systems of left-linear AC completion allow for more efficient joinability checks using $\to_{\mathcal{R}}$ instead of $\to_{\mathcal{R}/\mathsf{AC}}$. We leave this for future work.

The complete results are available on the tool's website[4]. We conclude with some additional notes on the results.

– The results are not cluttered with detailed results for the available options regarding prime critical pairs and the concrete rewrite relation used for simplify and compose since they did not lead to significant runtime differences. Instead, the default options (no prime critical pairs and the rewrite relation $\to_{\mathcal{R}}$) were used for the experiments.
– The second problem in Table 1 shows the merits of using termination tools as it includes round-up division which cannot be handled by simplification orders.
– Due to the incompleteness of the used approach for completion with termination tools, some equations in the problems `A95_ex4_2_4a.trs` as well as `sp.trs` had to be reversed in order to get appropriate results. Note that this does not distort the experimental results for left-linear AC completion in general as the problem lies in the particular implementation of completion with termination tools.

8 Conclusion

In this paper, we consolidated the existing literature for left-linear AC completion in the case of finite runs and gave new insight into its merits compared to general AC completion. Furthermore, our implementation accompll allowed us to run practical experiments. An extended version of this paper with full proof details and an appendix which describes the original inference systems of Avenhaus and Bachmair is available on the website of accompll (see Footnote 4). We conclude by giving some pointers for future work. First of all, the merits of our novel simulation result for general AC completion could be evaluated experimentally by providing an implementation. Another interesting research direction is normalized completion for the left-linear case. If successful, this would facilitate the treatment of important cases such as abelian groups despite the restriction to left-linear TRSs. Furthermore, a formalization of the established theoretical results is desirable. To that end, the existing Isabelle/HOL formalization from [4] is a perfect starting point as some results of this paper are extensions of the results for standard rewriting presented there.

Acknowledgments. We thank Jonas Schöpf and Fabian Mitterwallner for providing the experimental version of $\mathsf{T_TT_2}$ as well as the anonymous reviewers for their valuable suggestions.

[4] http://cl-informatik.uibk.ac.at/software/accompll/.

References

1. Avenhaus, J.: Reduktionssysteme. Springer, Heidelberg (1995). https://doi.org/10.1007/978-3-642-79351-6. (in German)
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998). https://doi.org/10.1017/CBO9781139172752
3. Bachmair, L.: Canonical Equational Proofs. Progress in Theoretical Computer Science. Birkhäuser, Boston (1991). https://doi.org/10.1007/978-1-4684-7118-2
4. Hirokawa, N., Middeldorp, A., Sternagel, C., Winkler, S.: Abstract completion, formalized. Logical Methods Comput. Sci. **15**(3), 19:1–19:42 (2019). https://doi.org/10.23638/LMCS-15(3:19)2019
5. Huet, G.: Confluent reductions: abstract properties and applications to term rewriting systems. J. ACM **27**(4), 797–821 (1980). https://doi.org/10.1145/322217.322230
6. Jouannaud, J.P., Kirchner, H.: Completion of a set of rules modulo a set of equations. SIAM J. Comput. **15**(4), 1155–1194 (1986). https://doi.org/10.1137/0215084
7. Kapur, D.: A modular associative commutative (AC) congruence closure algorithm. In: Kobayashi, N. (ed.) Proceedings of the 6th International Conference on Formal Structures for Computation and Deduction. Leibniz International Proceedings in Informatics, vol. 195, pp. 15:1–15:21 (2021). https://doi.org/10.4230/LIPIcs.FSCD.2021.15
8. Kapur, D., Musser, D.R., Narendran, P.: Only prime superpositions need be considered in the Knuth-Bendix completion procedure. J. Symb. Comput. **6**(1), 19–36 (1988). https://doi.org/10.1016/S0747-7171(88)80019-1
9. Klein, D., Hirokawa, N.: Maximal completion. In: Schmidt-Schauß, M. (ed.) Proceedings of the 22nd International Conference on Rewriting Techniques and Applications. Leibniz International Proceedings in Informatics, vol. 10, pp. 71–80 (2011). https://doi.org/10.4230/LIPIcs.RTA.2011.71
10. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press (1970). https://doi.org/10.1016/B978-0-08-012975-4.50028-X
11. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean Termination Tool 2. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 295–304. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02348-4_21
12. Marché, C.: Normalized rewriting: an alternative to rewriting modulo a set of equations. J. Symb. Comput. **21**(3), 253–288 (1996). https://doi.org/10.1006/jsco.1996.0011
13. Sternagel, C., Thiemann, R.: Formalizing Knuth-Bendix orders and Knuth-Bendix completion. In: Raamsdonk, F. (ed.) Proceedings of the 24th International Conference on Rewriting Techniques and Applications. Leibniz International Proceedings in Informatics, vol. 21, pp. 286–301 (2013). https://doi.org/10.4230/LIPIcs.RTA.2013.287
14. Sternagel, T., Zankl, H.: KBCV – Knuth-Bendix completion visualizer. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 530–536. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_41
15. Wehrman, I., Stump, A., Westbrook, E.: Slothrop: Knuth-Bendix completion with a modern termination checker. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 287–296. Springer, Heidelberg (2006). https://doi.org/10.1007/11805618_22

16. Winkler, S.: Termination tools in automated reasoning. Ph.D. thesis, University of Innsbruck (2013)
17. Winkler, S.: Extending maximal completion. In: Geuvers, H. (ed.) Proceedings of the 4th International Conference on Formal Structures for Computation and Deduction. Leibniz International Proceedings in Informatics, vol. 131, pp. 3:1–3:15 (2019). https://doi.org/10.4230/LIPIcs.FSCD.2019.3
18. Winkler, S., Middeldorp, A.: AC completion with termination tools. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 492–498. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_37
19. Winkler, S., Middeldorp, A.: Normalized completion revisited. In: Raamsdonk, F. (ed.) Proceedings of the 24th International Conference on Rewriting Techniques and Applications. Leibniz International Proceedings in Informatics, vol. 21, pp. 318–333 (2013). https://doi.org/10.4230/LIPIcs.RTA.2013.319
20. Winkler, S., Sato, H., Middeldorp, A., Kurihara, M.: Multi-completion with termination tools. J. Autom. Reason. **50**(3), 317–354 (2013). https://doi.org/10.1007/s10817-012-9249-2
21. Yamada, A., Kusakari, K., Sakabe, T.: Nagoya Termination Tool. In: Dowek, G. (ed.) RTA 2014. LNCS, vol. 8560, pp. 466–475. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_32

On P-Interpolation in Local Theory Extensions and Applications to the Study of Interpolation in the Description Logics $\mathcal{EL}, \mathcal{EL}^+$

Dennis Peuter, Viorica Sofronie-Stokkermans(✉), and Sebastian Thunert

University of Koblenz, Koblenz, Germany
{dpeuter,sofronie}@uni-koblenz.de, S.Thunert@gmx.de

Abstract. We study the P-interpolation property for certain local theory extensions, and use these results for proving $\leq$-interpolation in classes of semilattices with monotone operators. For computing the $\leq$-interpolating terms, we use a hierarchic approach. We use these results for the study of $\sqsubseteq$-interpolation in the description logics $\mathcal{EL}$ and $\mathcal{EL}^+$.

1 Introduction

In this paper we study the problem of P-interpolation, a problem strongly related to interpolation w.r.t. logical theories. The problem can be formulated as follows:

Let $\mathcal{T}$ be a theory, A and B be conjunctions of ground literals in the signature of $\mathcal{T}$, possibly with additional constants, P a predicate symbol in the signature of $\mathcal{T}$, a a constant occurring in A and b a constant occurring in B. Assume that $A \wedge B \models_{\mathcal{T}} aPb$. Can we find a ground term t containing only constants and function symbols "shared" by A and B, such that $A \wedge B \models_{\mathcal{T}} aPt \ \wedge \ tPb$?

Interpolation has been studied in classical and non-classical logics and in extensions and combinations of theories; and is very important in program verification and also in the area of description logics. The first algorithms for interpolant generation in program verification required explicit constructions and "separations" of proofs [14,16]. In [13] interpolants are computed using variants of resolution. For certain theories, the "separation" of proofs relied on the possibility of "separating" atoms, i.e. on P-interpolation. Equality interpolation is used in [34] for devising an interpolation method in combinations of theories with disjoint signatures. In [22,24] and [19], for instance, we consider interpolation problems in certain classes of extensions $\mathcal{T}_0 \cup \mathcal{K}$ of a base theory $\mathcal{T}_0$ and use a hierarchical approach to compute interpolants. The method relies on the P-interpolation property of the base theory $\mathcal{T}_0$. In most of the applications we considered, P is the equality predicate $\approx$ or a predicate $\leq$ with the property that in all models of $\mathcal{T}_0$, the interpretation of $\leq$ is a partial ordering. Since at that time our main interest was the study of *interpolation problems*, in [22,24] and [19] P-interpolation is only used in order to help in giving methods for interpolation and not as a goal in itself. However, in several papers in the area

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 419–437, 2023.
https://doi.org/10.1007/978-3-031-38499-8_24

of description logics (cf. e.g. [8,31]) when defining the notion of interpolation in description logics the authors define in fact a notion of $\sqsubseteq$-interpolation. In [8] (Thm. 4) it is proved that $\mathcal{EL}^+$ allows interpolation (in fact, the notion of $\sqsubseteq$-interpolation mentioned above) for *safe* role inclusions – this is related to the notion of "sharing" considered in [24], cf. also Sect. 4. The proof technique in [8] uses simulations. In this paper, we analyze the property of P-interpolation in theory extensions, propose a method for solving it based on hierarchical reasoning and satisfiability modulo theories, and formulate the $\sqsubseteq$-interpolation problem for $\mathcal{EL}$ and $\mathcal{EL}^+$ as a $\leq$-interpolation problem in a theory of semilattices with operators. We first studied $\leq$-interpolation in [17] in the context of description logics; the $\sqsubseteq$-interpolating concept descriptions were regarded as a form of "high-level" explanations. In this paper we further extend the work in [17]. The general approach we propose opens the possibility of applying similar methods to more general classes of non-classical logics (including e.g. substructural logics or the logics with monotone operators studied in [27,28]) or in verification (to consider more general theory extensions than those with uninterpreted function symbols analyzed in [19]). The main results can be summarized as follows:

- We propose variants of the definitions of convexity, P-interpolation and Beth definability relative to a subsignature.
- We describe a hierarchical P-interpolation method in certain classes of local theory extensions.
- We illustrate the applicability of these results to prove that certain classes of semilattices with monotone operators have the property of $\leq$-interpolation for a certain interpretation of "shared" function symbols.
- We show, by giving a counterexample, that $\leq$-interpolation does not hold if by "shared" symbols we mean just the *common* symbols.
- We indicate how these results can be used to prove or disprove various notions of interpolation for the description logics $\mathcal{EL}$ and $\mathcal{EL}^+$.

Structure of the Paper: In Sect. 2 and 3 basic notions are introduced, and some results needed later are proved. In Sect. 4 we identify classes of local theory extensions allowing P-interpolation and propose a hierarchical method of computing P-interpolants. This is used in Sect. 5 to study the existence of $\leq$-interpolation in classes of semilattices with monotone operators. In Sect. 6 we use the links between the theory of semilattices with operators and the description logics $\mathcal{EL}$ and $\mathcal{EL}^+$, and show how the results can be used in the study of these logics. The details of the proofs and additional examples can be found in [18].

2 Theories, Convexity, *P*-Interpolation, Beth Definability

We assume known standard definitions from first-order logic such as Π-structures, models, homomorphisms, logical entailment, satisfiability, unsatisfiability.

We consider signatures of the form $\Pi = (\Sigma, \mathsf{Pred})$, where Σ is a family of function symbols and Pred a family of predicate symbols. In this paper, a theory

$\mathcal{T}$ is described by a set of closed formulae (the axioms of the theory). We call a theory axiomatized by a set of (universally quantified) equations an *equational theory*. In this paper, we denote by $\mathsf{Mod}(\mathcal{T})$ the set of all models of $\mathcal{T}$. We denote "falsum" with $\perp$. If F and G are formulae we write $F \models G$ (resp. $F \models_{\mathcal{T}} G$) to express the fact that every model of F (resp. every model of F which is also a model of $\mathcal{T}$) is a model of G. The definitions can be extended in a natural way to the case when F is a set of formulae; in this case, $F \models_{\mathcal{T}} G$ if and only if $\mathcal{T} \cup F \models G$. $F \models \perp$ means that F is unsatisfiable; $F \models_{\mathcal{T}} \perp$ means that there is no model of $\mathcal{T}$ which is also a model of F. If there is a model of $\mathcal{T}$ which is also a model of F we say that F is $\mathcal{T}$-consistent. If C is a fixed countable set of fresh constants, we denote by Π^C the extension of Π with constants in C.

Convexity and P-Convexity. We can define a notion of convexity w.r.t. a subset P of the set of predicates.

Definition 1. *A theory $\mathcal{T}$ with signature $\Pi = (\Sigma, \mathsf{Pred})$ is* convex *with respect to a subset P of* Pred *(which may include also equality $\approx$) if for all conjunctions Γ of ground Π^C-atoms (with additional constants in a set C), relations $R_1, \dots, R_m \in P$ and tuples of Π^C-terms of corresponding arity $\overline{t}_1, \dots, \overline{t}_m$ such that $\Gamma \models_{\mathcal{T}} \bigvee_{i=1}^m R_i(\overline{t}_i)$ there exists $i_0 \in \{1, \dots, m\}$ such that $\Gamma \models_{\mathcal{T}} R_{i_0}(\overline{t}_{i_0})$.*

We will call a theory $\mathcal{T}$ *convex* if it is $\mathsf{Pred} \cup \{\approx\}$-convex. The following result is well-known (cf. e.g. [5,10,32]):

Theorem 1. *Let $\mathcal{T}$ be a theory and let $\mathsf{Mod}(\mathcal{T})$ be the class of models of $\mathcal{T}$.*

(i) If $\mathsf{Mod}(\mathcal{T})$ is closed under direct products then $\mathcal{T}$ is convex.
(ii) If $\mathcal{T}$ is a universal theory and $\mathcal{T}$ is convex, then $\mathcal{T}$ has an axiomatization given by Horn clauses, hence $\mathsf{Mod}(\mathcal{T})$ is closed under direct products.

Corollary 2. *Let $\mathcal{T}_1$, $\mathcal{T}_2$ be two theories with signatures Π_1, Π_2. If $\mathsf{Mod}(\mathcal{T}_1)$ and $\mathsf{Mod}(\mathcal{T}_2)$ are closed under direct products, then $\mathcal{T}_1 \cup \mathcal{T}_2$ is convex.*

Proof: Follows from the fact that if $\mathsf{Mod}(\mathcal{T}_1)$ and $\mathsf{Mod}(\mathcal{T}_2)$ are closed under direct products then so is also $\mathsf{Mod}(\mathcal{T}_1 \cup \mathcal{T}_2)$ and from Theorem 1. □

From Theorem 1 and Corollary 2 it immediately follows that if $\mathcal{T}_1$ and $\mathcal{T}_2$ are universal theories and convex then $\mathcal{T}_1 \cup \mathcal{T}_2$ is convex. In particular, every extension of a convex universal theory $\mathcal{T}_0$ with a set of new function symbols axiomatized by a set $\mathcal{K}$ of Horn clauses is convex.

Equality Interpolation, R-Interpolation. We say that a convex theory $\mathcal{T}$ has the equality interpolation property if for every conjunction of ground Π^C-literals $A(\overline{c}, \overline{a_1}, a)$ and $B(\overline{c}, \overline{b_1}, b)$, if $A \wedge B \models_{\mathcal{T}} a \approx b$ then there exists a term $t(\overline{c})$ containing only the shared constants $\overline{c}$ such that $A \wedge B \models_{\mathcal{T}} a \approx t(\overline{c}) \ \wedge \ t(\overline{c}) \approx b$.

Sometimes, the theories and theory extensions we study contain interpreted symbols in a set $\Pi_0 = (\Sigma_0, \mathsf{Pred})$ and non-interpreted function symbols in a set Σ_1. The classical definition for equality interpolation for a theory $\mathcal{T}$ mentioned above allows the term $t(\overline{c})$ to contain all function symbols in the signature of $\mathcal{T}$

– these symbols are in this case all seen as being interpreted. If we distinguish between interpreted and uninterpreted functions we might require that the intermediate term $t(\overline{c})$ contains only "shared" uninterpreted functions and common constants.

If Σ_A and Σ_B are the uninterpreted function symbols occurring in A resp. B, and Θ is a closure operator, by "shared" uninterpreted functions we can mean:

- *Intersection-shared symbols*: $\bigcap\text{-}\mathsf{Shared}(A, B) = \Sigma_A \cap \Sigma_B$, or
- *Θ-shared symbols*: $\Theta\text{-}\mathsf{Shared}(A, B) = \Theta(\Sigma_A) \cap \Theta(\Sigma_B)$.

Example 1. Let $\mathcal{T} = \mathcal{T}_0 \cup \mathcal{K}$ be the extension of a theory $\mathcal{T}_0$ with set of interpreted function symbols Σ_0 with a set $\mathcal{K}$ of clauses containing new uninterpreted function symbols in a set Σ_1. If A and B are sets of atoms in the signature of $\mathcal{T}$ containing additional constants in a set C and uninterpreted function symbols Σ_A, Σ_B then the *intersection-shared* uninterpreted function symbols of A and B are $\Sigma_A \cap \Sigma_B$. Let $\Theta_{\mathcal{K}}$ be defined for every $\Sigma \subseteq \Sigma_1$ by $\Theta_{\mathcal{K}}(\Sigma) = \bigcup_{f \in \Sigma}\{g \in \Sigma_1 \mid g \sim^*_{\mathcal{K}} f\}$, where $\sim^*_{\mathcal{K}}$ is the equivalence relation induced by the relation $f \sim_{\mathcal{K}} g$ iff there exists $C{\in}\mathcal{K}$ s.t. f, g both occur in C.
Then the $\Theta_{\mathcal{K}}$-shared symbols are $\Theta_{\mathcal{K}}(\Sigma_A) \cap \Theta_{\mathcal{K}}(\Sigma_B)$. In particular, if A contains a function symbol f and B contains a symbol g such that f, g occur both in a clause in $\mathcal{K}$, then f and g are considered to be $\Theta_{\mathcal{K}}$-shared by A, B. □

We also might be interested in similar properties for other binary relations. We define an R-interpolation property, where R is a binary predicate symbol in Π.

Definition 2. *Let $R \in \mathsf{Pred} \cup \{\approx\}$ be a binary predicate symbol. An $\{R\}$-convex theory $\mathcal{T}$ with uninterpreted symbols Σ_1 has the* R-interpolation property *if for all conjunctions of ground atoms $A(\overline{c}, \overline{a_1}, a)$ and $B(\overline{c}, \overline{b_1}, b)$, if $A \wedge B \models_{\mathcal{T}} aRb$ then there exists a term $t(\overline{c})$ containing only common constants $\overline{c}$ and only "shared" uninterpreted symbols in Σ_1 such that $A \wedge B \models_{\mathcal{T}} aRt(\overline{c}) \ \wedge \ t(\overline{c})Rb$.*

If $P \subseteq \mathsf{Pred}$, we say that a theory has the *P-interpolation property* if it has the R-interpolation property for every $R \in P$. In Sect. 5 we give examples of theories with this property and show that a theory may not have the R-interpolation property for a predicate symbol R if we use the notion of *intersection-shared symbols*, but has the R-interpolation property if we consider the less restrictive notion of *Θ-shared symbols* for a suitably defined closure operator Θ.

Beth Definability. Let $\mathcal{T}$ be a theory with signature $\Pi = (\Sigma_0 \cup \Sigma_1, \mathsf{Pred})$, where the function symbols in Σ_0 are interpreted function symbols and the function symbols in Σ_1 are regarded as uninterpreted function symbols, and let C be a set of additional constants. We define a notion of Beth definability relative to a subset $\Sigma_S \subseteq \Sigma_1 \cup C$ of non-interpreted function symbols and constants similar to the one introduced in [31], which we refer to as Σ_S-Beth definability.
Let $\Sigma_S \subseteq \Sigma_1 \cup C$, let $\Sigma_r = \Sigma_1 \backslash \Sigma_S$, and let $\Pi' = (\Sigma_0 \cup (\Sigma_S \cap \Sigma_1) \cup \Sigma'_r, \mathsf{Pred})$, where $\Sigma'_r = \{f' \mid f \in \Sigma_1 \backslash \Sigma_S\}$ is the signature obtained by replacing all uninterpreted function symbols in Σ_1 which are not in Σ_S with new primed copies. If ϕ is a Π^C-formula, we will denote by ϕ' the formula obtained from ϕ by replacing

all uninterpreted function symbols in $\Sigma_1 \backslash \Sigma_S$ and all constants in $C \backslash \Sigma_S$ with distinct, primed versions. The interpreted function symbols and the uninterpreted function symbols and constants in Σ_S are not changed. We regard the theory $\mathcal{T}$ as a set of formulae; let $\mathcal{T}' := \{\phi' \mid \phi \in \mathcal{T}\}$.[1]

Let A be a conjunction of ground Π^C-literals, and $a \in C$. We say that a is *implicitly defined* by A w.r.t. Σ_S and $\mathcal{T}$ if, with the notations introduced before,

$$A \wedge A' \models_{\mathcal{T} \cup \mathcal{T}'} a \approx a'.$$

We say that a is *explicitly defined* by A w.r.t. Σ_S and $\mathcal{T}$ if there exists a term t containing only symbols in Σ_0, Pred and Σ_S such that $A \models_{\mathcal{T}} a \approx t$.

Definition 3. *Let $\mathcal{T}$ be a theory with uninterpreted function symbols in a set Σ_1. Let $\Sigma_S \subseteq \Sigma_1 \cup C$. $\mathcal{T}$ has the* Beth definability property w.r.t. Σ_S *(Σ_S-Beth definability), if for every conjunction of literals A and every $a \in C$, if A implicitly defines a w.r.t. Σ_S and $\mathcal{T}$ then A explicitly defines a w.r.t. Σ_S and $\mathcal{T}$.*

In [4,6] it was proved that if a convex theory has the $\approx$-interpolation property, then it has the Beth definability property. We give an analogous implication between $\approx$-interpolation and Beth definability w.r.t. a subsignature.

Theorem 3. *Let $\mathcal{T}$ be a convex theory with signature $\Pi = (\Sigma_0 \cup \Sigma_1, \mathsf{Pred})$, C a set of constants, and $\Sigma_S \subseteq \Sigma_1 \cup C$. Let $\mathcal{T}'$ be as defined above.*

(i) If $\mathcal{T} \cup \mathcal{T}'$ has the $\approx$-interpolation property with intersection-sharing, then $\mathcal{T}$ has the Σ_S-Beth definability property.
(ii) Assume that $\mathcal{T} = \mathcal{T}_0 \cup \mathcal{K}$ where all symbols in the signature of $\mathcal{T}_0$ are regarded as interpreted, and $\mathcal{K}$ is a set of clauses also containing uninterpreted function symbols in Σ_1. Let $\Theta_{\mathcal{K}}$ be the closure operator defined in Example 1. If $\mathcal{T} \cup \mathcal{T}'$ has the $\approx$-interpolation property with $\Theta_{\mathcal{K} \cup \mathcal{K}'}$-sharing, then $\mathcal{T}$ has the $\Theta_{\mathcal{K}}(\Sigma_S)$-Beth definability property.

Proof (Idea): (i) Assume a is implicitly definable w.r.t. Σ_S, i.e. there exists a conjunction A of literals such that if A' is obtained by renaming as explained before, then $A \wedge A' \models_{\mathcal{T} \cup \mathcal{T}'} a \approx a'$. Since $\mathcal{T} \wedge \mathcal{T}'$ has $\approx$-interpolation, there exists a term t using only the functions and predicate symbols common to A and A' (i.e. the symbols in $\Sigma_0 \cup \Sigma_S$) such that $A \wedge A' \models_{\mathcal{T} \cup \mathcal{T}'} a \approx t \wedge t \approx a'$. It can be shown that then $A \models_{\mathcal{T}} a \approx t$.

(ii) Assume a is implicitly definable w.r.t. $\Theta_{\mathcal{K}}(\Sigma_S)$, i.e. there exists a conjunction A of literals such that if A' is obtained by renaming as explained before then $A \wedge A' \models_{\mathcal{T} \cup \mathcal{T}'} a \approx a'$. The symbols shared by A and A' are the symbols in $\Sigma_0 \cup \Sigma_S \cup \Theta_{\mathcal{K} \cup \mathcal{K}'}(\Sigma_S)$, where $\Theta_{\mathcal{K} \cup \mathcal{K}'}(\Sigma_S) = \bigcup_{f \in \Sigma_S \cap \Sigma_1} \{g \in \Sigma_1 \cup \Sigma_1' \mid f \sim^*_{\mathcal{K} \cup \mathcal{K}'} g\}$. It is easy to see that for every $f \in \Sigma_1 \backslash \Sigma_S$, $f \in \Theta_{\mathcal{K}}(\Sigma_S)$ iff $f' \in \Theta_{\mathcal{K}'}(\Sigma_S)$, and $\Theta_{\mathcal{K} \cup \mathcal{K}'}(\Sigma_S) = \Theta_{\mathcal{K}}(\Sigma_S) \cup \Theta_{\mathcal{K}'}(\Sigma_S)$. Since we assumed that $\mathcal{T} \cup \mathcal{T}'$ has the $\approx$-interpolation property with the notion of $\Theta_{\mathcal{K} \cup \mathcal{K}'}$-sharing, there exists a term t over the signature $\Sigma_0 \cup \Theta_{\mathcal{K} \cup \mathcal{K}'}(\Sigma_S)$ such that $A \wedge A' \models_{\mathcal{T} \cup \mathcal{T}'} a \approx t \ \wedge \ t \approx a$. The term t might contain primed versions of function symbols. We can show that we can find a term $\bar{t}$ containing only terms in $\Theta_{\mathcal{K}}(\Sigma_S)$ such that $A \models_{\mathcal{T}} a \approx \bar{t}$. □

[1] A similar definition can be given if theories are defined as classes of models.

3 Local Theory Extensions

Let $\Pi_0{=}(\Sigma_0, \mathsf{Pred})$ be a signature, and $\mathcal{T}_0$ be a "base" theory with signature Π_0. We consider extensions $\mathcal{T} := \mathcal{T}_0 \cup \mathcal{K}$ of $\mathcal{T}_0$ with new function symbols Σ_1 (*extension functions*) whose properties are axiomatized using a set $\mathcal{K}$ of (universally closed) clauses in the extended signature $\Pi = (\Sigma_0 \cup \Sigma_1, \mathsf{Pred})$, which contain function symbols in Σ_1. If G is a finite set of ground Π^C-clauses, where C is an additional set of constants, and $\mathcal{K}$ a set of Π-clauses, we will denote by $\mathsf{st}(\mathcal{K}, G)$ (resp. $\mathsf{est}(\mathcal{K}, G)$) the set of all ground terms (resp. extension ground terms, i.e. terms starting with a function in Σ_1) which occur in G or $\mathcal{K}$. In this paper we regard every finite set G of ground clauses as the ground formula $\bigwedge_{C \in G} C$. If T is a set of ground terms in the signature Π^C, we denote by $\mathcal{K}[T]$ the set of all instances of $\mathcal{K}$ in which the terms starting with a function symbol in Σ_1 are in T. Let Ψ be a map associating with every finite set T of ground terms a finite set $\Psi(T)$ of ground terms containing T. For any set G of ground Π^C-clauses we write $\mathcal{K}[\Psi_{\mathcal{K}}(G)]$ for $\mathcal{K}[\Psi(\mathsf{est}(\mathcal{K}, G))]$. We define:

(Loc_f^{Ψ}) For every finite set G of ground clauses in Π^C it holds that
$\mathcal{T}_0 \cup \mathcal{K} \cup G \models \perp$ if and only if $\mathcal{T}_0 \cup \mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G$ is unsatisfiable.

Extensions satisfying condition (Loc_f^{Ψ}) are called *Ψ-local.* If Ψ is the identity we obtain the notion of *local theory extensions* [21]; if in addition $\mathcal{T}_0$ is the theory of pure equality we obtain the notion of *local theories* [9,15].

Hierarchical Reasoning. Consider a Ψ-local theory extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$. Condition ($\mathsf{Loc}_f^{\Psi}$) requires that for every finite set G of ground Π^C-clauses, $\mathcal{T}_0{\cup}\mathcal{K}{\cup}G \models\perp$ iff $\mathcal{T}_0{\cup}\mathcal{K}[\Psi_{\mathcal{K}}(G)]{\cup}G \models\perp$. In all clauses in $\mathcal{K}[\Psi_{\mathcal{K}}(G)]{\cup}G$ the function symbols in Σ_1 only have ground terms as arguments, so $\mathcal{K}[\Psi_{\mathcal{K}}(G)]{\cup}G$ can be flattened and purified by introducing, in a bottom-up manner, new constants $c_t \in C$ for subterms $t{=}f(c_1, \dots, c_n)$ where $f{\in}\Sigma_1$ and c_i are constants, together with definitions $c_t{=}f(c_1, \dots, c_n)$. We thus obtain a set of clauses $\mathcal{K}_0{\cup}G_0{\cup}\mathsf{Def}$, where $\mathcal{K}_0$ and G_0 do not contain Σ_1-function symbols and Def contains clauses of the form $c{=}f(c_1, \dots, c_n)$, where $f{\in}\Sigma_1$, $c, c_1, \dots, c_n$ are constants.

Theorem 4 ([11,12,21]). *Let $\mathcal{K}$ be a set of clauses. Assume that $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$ is a Ψ-local theory extension. For any finite set G of flat ground clauses (with no nestings of extension functions), let $\mathcal{K}_0 \cup G_0 \cup \mathsf{Def}$ be obtained from $\mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G$ by flattening and purification, as explained above. Then the following are equivalent to $\mathcal{T}_0 \cup \mathcal{K} \cup G \models\perp$:*

(i) $\mathcal{T}_0{\cup}\mathcal{K}[\Psi_{\mathcal{K}}(G)]{\cup}G \models\perp$.
(ii) $\mathcal{T}_0 \cup \mathcal{K}_0 \cup G_0 \cup \mathsf{Con}_0 \models\perp$, *where*

$$\mathsf{Con}_0 = \left\{ \bigwedge_{i=1}^{n} c_i \approx d_i \to c \approx d \;\middle|\; \begin{matrix} f(c_1, \dots, c_n) \approx c \in \mathsf{Def} \\ f(d_1, \dots, d_n) \approx d \in \mathsf{Def} \end{matrix} \right\}.$$

In [12] we showed that for extensions with sets of flat and linear clauses Ψ-locality can be checked by checking whether an embeddability condition of partial into total models holds.In [26] we mention (without proof) that the proof in [12] can be extended to situations in which the clauses in $\mathcal{K}$ are not linear. The result is presented below. A full proof is given in the extended version of this paper [18].

Theorem 5. *Let $\mathcal{K}$ be a set of Σ_1-flat clauses, and $\Psi_{\mathcal{K}}$ be a term closure operator such that for every set T of ground terms and for every clause D in $\mathcal{K}$, if a variable occurs in two terms in D then either the two terms are identical, or the variable occurs below two different unary function symbols f and g and, for every constant c, $f(c)$ is in $\Psi(T)$ iff $g(c)$ is in $\Psi(T)$. If all partial models $\mathcal{A}$ of $\mathcal{T}_0 \cup \mathcal{K}$ with totally defined Σ_0-functions, and for which the set of terms $\{f(a_1, \dots, a_n) \mid f \in \Sigma_1$ and $f_{\mathcal{A}}(a_1, \dots, a_n)$ is defined$\}$ is finite and closed under Ψ, embed into total models of $\mathcal{T}_0 \cup \mathcal{K}$, then the extension $\mathcal{T}_0 \cup \mathcal{K}$ satisfies (Loc_f^{Ψ}).*

4 *R*-interpolation in Local Theory Extensions

In [24] we considered convex and P-interpolating theories $\mathcal{T}_0$ with signature $\Pi_0 = (\Sigma_0, \mathsf{Pred})$ (where $P \subseteq \mathsf{Pred}$). We studied Ψ-local extensions $\mathcal{T} = \mathcal{T}_0 \cup \mathcal{K}$ of $\mathcal{T}_0$ with new function symbols in a set Σ_1 axiomatized by a set $\mathcal{K}$ of clauses, with the property that all clauses in $\mathcal{K}$ are of the form:

$$\begin{cases} x_1\, R_1\, s_1 \wedge \dots \wedge x_n\, R_n\, s_n \rightarrow f(x_1, \dots, x_n)\, R\, g(y_1, \dots, y_n) \\ x_1\, R_1\, y_1 \wedge \dots \wedge x_n\, R_n\, y_n \rightarrow f(x_1, \dots, x_n)\, R\, f(y_1, \dots, y_n) \end{cases} \tag{1}$$

where $n \geq 1$, $x_1, \dots, x_n, y_1, \dots, y_n$ are variables, $f, g \in \Sigma_1$, $R_1, \dots, R_n, R$ are binary relations with $R_1, \dots, R_n \in P$ and R transitive, and each s_i is either a variable among the arguments of g, or a term of the form $f_i(z_1, \dots, z_k)$, where $f_i \in \Sigma_1$ and all the arguments of f_i are variables occurring among the arguments of g.

Example 2. A set $\mathcal{K}$ of axioms containing clauses of the form:

$$\begin{cases} x_1 \leq h(y_1) \rightarrow f(x_1) \leq g(y_1) \\ \quad x_1 \leq y_1 \rightarrow f(x_1) \leq f(y_1) \end{cases}$$

satisfies the conditions above: $n = 1$, $R_1 = R = \leq$, $s_1 = h(y_1)$, $f, g, h \in \Sigma_1$.

In [24], we proved that if $\mathcal{T}_0$ allows ground interpolation, then $\mathcal{T}$ allows ground interpolation, and that the interpolants can be computed in a hierarchical way, using a method for ground interpolation in $\mathcal{T}_0$. We now show that under the conditions above, the property of *P-interpolation* can be transferred from the theory $\mathcal{T}_0$ to the extension $\mathcal{T} = \mathcal{T}_0 \cup \mathcal{K}$ of $\mathcal{T}_0$. The function symbols in the signature of $\mathcal{T}_0$ are considered to be interpreted, and will always be considered to be shared. For the function symbols in the signature Σ_1 – considered to be "quasi"-interpreted – we use the notion of $\Theta_{\mathcal{K}}$-sharing introduced in Sect. 2.

In order to show that $\mathcal{T}$ has the P-interpolation property, we need to prove that if A, B are conjunctions of atoms and $A(\overline{c}, \overline{a}_1, a) \wedge B(\overline{c}, \overline{b}_1, b) \models_{\mathcal{T}} aRb$, where $R \in P$, then there exists a term t containing only the constants common to A

and B and only function symbols which are $\Theta_{\mathcal{K}}$*-shared* by A and B, such that $A(\overline{c},\overline{a}_1,a)\wedge B(\overline{c},\overline{b}_1,b)\models_{\mathcal{T}} aRt \ \wedge\ tRb$.

$A(\overline{c},\overline{a}_1,a)\wedge B(\overline{c},\overline{b}_1,b)\models_{\mathcal{T}} aRb$ iff $A(\overline{c},\overline{a}_1,a)\wedge B(\overline{c},\overline{b}_1,b)\wedge\neg(aRb) \models_{\mathcal{T}} \perp$. By Theorem 4 we can purify and flatten this conjunction and obtain a conjunction of unit clauses $A_0\wedge B_0\wedge \mathsf{Def}\wedge\neg(aRb)$, where Def is a set of definitions of newly introduced constants. Let T be the extension terms in Def. We introduce new constants and definitions also for all extension terms in $\Psi(T)$. This new set of definitions can be written as a conjunction $D_A\wedge D_B$ of its A-part and its B-part. By the Ψ-locality of the extension $\mathcal{T}_0\subseteq\mathcal{T}_0\cup\mathcal{K}$ and Theorem 4,

$$A_0\wedge B_0\wedge\mathsf{Def}\wedge\neg(aRb)\models_{\mathcal{T}}\perp \text{ iff } \mathcal{K}_0\wedge A_0\wedge B_0\wedge\mathsf{Con}[D_A\wedge D_B]_0\wedge\neg(aRb)\models_{\mathcal{T}_0}\perp,$$

where $\mathcal{K}_0$ is obtained from $\mathcal{K}[D_A\wedge D_B]$ by replacing the Σ_1-terms with the corresponding constants contained in the definitions $D_A\wedge D_B$ and

$$\mathsf{Con}[D_A\wedge D_B]_0=\bigwedge\left\{\bigwedge_{i=1}^{n} c_i\approx d_i\to c\approx d \mid \begin{array}{l} f(c_1,\dots,c_n)\approx c\in D_A\cup D_B,\\ f(d_1,\dots,d_n)\approx d\in D_A\cup D_B\end{array}\right\}.$$

In general, $\mathsf{Con}[D_A\wedge D_B]_0=\mathsf{Con}_0^A\wedge\mathsf{Con}_0^B\wedge\mathsf{Con}_{\mathsf{mix}}$ and $\mathcal{K}_0=\mathcal{K}_0^A\wedge\mathcal{K}_0^B\wedge\mathcal{K}_{\mathsf{mix}}$, where $\mathsf{Con}_0^A,\mathcal{K}_0^A$ only contain extension functions and constants which occur in A, $\mathsf{Con}_0^B,\mathcal{K}_0^B$ only contain extension functions and constants which occur in B, and $\mathsf{Con}_{\mathsf{mix}}$, $\mathcal{K}_{\mathsf{mix}}$ contain mixed clauses with constants occurring in both A and B. Our goal is to separate $\mathsf{Con}_{\mathsf{mix}}$ and $\mathcal{K}_{\mathsf{mix}}$ into an A-part and a B-part, which would allow us to use the P-interpolation property of theory $\mathcal{T}_0$.

Proposition 6. *Assume that $\mathcal{T}_0$ is convex and P-interpolating. Let $\mathcal{H}$ be a set of Horn clauses $(\bigwedge_{i=1}^n c_iR_id_i)\to cR_0d$ in the signature Π_0^C (with R_0 transitive and $R_i\in P$) which are instances of flattened and purified clauses of type (1) and of congruence axioms. Let $\mathcal{H}_{\mathsf{mix}}$ be the mixed clauses in $\mathcal{H}$:*

$$\begin{array}{l}\mathcal{H}_{\mathsf{mix}}=\{\bigwedge_{i=1}^n c_iR_id_i\to cR_0d\in\mathcal{H}\mid c_i,c \text{ constants in } A, d_i,d \text{ constants in } B\}\cup\\ \qquad\quad\ \{\bigwedge_{i=1}^n c_iR_id_i\to cR_0d\in\mathcal{H}\mid c_i,c \text{ constants in } B, d_i,d \text{ constants in } A\}\end{array}$$

Let A_0 and B_0 be conjunctions of ground literals in the signature Π_0^C such that $A_0\wedge B_0\wedge\mathcal{H}\wedge\neg(aRb)\models_{\mathcal{T}_0}\perp$. Then $\mathcal{H}$ can be separated into an A- and a B-part by replacing the set $\mathcal{H}_{\mathsf{mix}}$ of mixed clauses with a separated set of formulae $\mathcal{H}_{\mathsf{sep}}$:

(i) There exists a set T of $(\Sigma_0\cup C)$-terms containing only constants common to A_0 and B_0 such that $A_0\wedge B_0\wedge(\mathcal{H}\backslash\mathcal{H}_{\mathsf{mix}})\wedge\mathcal{H}_{\mathsf{sep}}\wedge\neg(aRb)\models_{\mathcal{T}_0}\perp$, where

$$\begin{array}{l}\mathcal{H}_{\mathsf{sep}}=\{(\bigwedge_{i=1}^n c_iR_it_i\to cRc_{f(t_1,\dots,t_n)})\wedge(\bigwedge_{i=1}^n t_iR_id_i\to c_{f(t_1,\dots,t_n)}Rd)\mid\\ \qquad\bigwedge_{i=1}^n c_iR_id_i\to cRd\in\mathcal{H}_{\mathsf{mix}}, d_i\approx s_i(e_1,\dots,e_n), d\approx g(e_1,\dots,e_n)\in D_B,\\ \qquad c\approx f(c_1,\dots,c_n)\in D_A \text{ or vice versa }\}=\mathcal{H}_{\mathsf{sep}}^A\wedge\mathcal{H}_{\mathsf{sep}}^B\end{array}$$

and $c_{f(t_1,\dots,t_n)}$ are new constants in Σ_c (considered to be common) introduced for the corresponding terms $f(t_1,\dots,t_n)$, where for $i\in\{1,\dots,n\}$, t_i separates the atom $c_iR_id_i$, which is entailed by the already deduced atoms.

(ii) $A_0\wedge B_0\wedge(\mathcal{H}\backslash\mathcal{H}_{\mathsf{mix}})\wedge\mathcal{H}_{\mathsf{sep}}\wedge\neg(aRb)$ is logically equivalent with respect to $\mathcal{T}_0$ with the following separated conjunction of ground literals:

$$\begin{array}{l}\overline{A}_0\wedge\overline{B}_0\wedge\neg(aRb)=A_0\ \wedge\ B_0\wedge\neg(aRb)\wedge\bigwedge\{cRd\mid\Gamma\to cRd\in\mathcal{H}\backslash\mathcal{H}_{\mathsf{mix}}\}\wedge\\ \qquad\qquad\bigwedge\{cRc_{f(\overline{t})}\wedge c_{f(\overline{t})}Rd\mid(\Gamma\to cRc_{f(\overline{t})})\wedge(\Gamma\to c_{f(\overline{t})}Rd)\in\mathcal{H}_{\mathsf{sep}}\}.\end{array}$$

Proof (Idea). The proof is similar to that of Prop. 5.7 in [24]. (i) and (ii) are proved simultaneously by induction on the number of clauses in $\mathcal{H}$. If $\mathcal{H} = \emptyset$, it is already separated. Otherwise, one can prove that either $(A_0 \wedge B_0) \models aRb$ – in which case we are done – or $A_0 \wedge B_0$ entails all the premises of some clause C in $\mathcal{H}$. If C contains only constants in A_0 or B_0 we can remove it from $\mathcal{H}$, add its conclusion to $A_0 \wedge B_0$ and repeat the procedure with the new $A_0 \wedge B_0$ and $\mathcal{H}$. If the clause is mixed, we can compute terms t_i which separate the premises in C, separate C into an instance C_1 of monotonicity and an instance C_2 of a clause in $\mathcal{K}$, remove C from $\mathcal{H}$, add to $A_0 \wedge B_0$ the conclusions of the clauses C_1, C_2, and repeat the procedure with the new $A_0 \wedge B_0$ and $\mathcal{H}$. □

Theorem 7. *Assume that $\mathcal{T}_0$ is convex and P-interpolating with respect to $P \subseteq \mathsf{Pred}$, and that $\mathcal{T} = \mathcal{T}_0 \cup \mathcal{K}$ is a local extension of $\mathcal{T}_0$ with a set of clauses $\mathcal{K}$ which only contains combinations of clauses of type (1). Then $\mathcal{T}$ is also P-interpolating.*

Proof (Idea). We prove that if A, B are conjunctions of literals and $A(\overline{c}, \overline{a}_1, a) \wedge B(\overline{c}, \overline{b}_1, b) \models_{\mathcal{T}} aRb$ where $R \in P$, then there exists a term t containing only the constants common to A and B and only function symbols which are shared by A and B, such that $A(\overline{c}, \overline{a}_1, a) \wedge B(\overline{c}, \overline{b}_1, b) \models_{\mathcal{T}} aRt \ \wedge \ tRb$. We can restrict w.l.o.g. to a purified and flattened conjunction of unit clauses $A_0 \wedge B_0 \wedge \mathsf{Def} \wedge \neg(aRb)$. With the notation used on page 8, by Theorem 4 we have:
$A_0 \wedge B_0 \wedge \mathsf{Def} \wedge \neg(aRb) \models_{\mathcal{T}} \perp$ iff $\mathcal{K}_0 \wedge A_0 \wedge B_0 \wedge \mathsf{Con}[D_A \wedge D_B]_0 \wedge \neg(aRb) \models_{\mathcal{T}_0} \perp$.
By Proposition 6 (ii), there exists a set T of $(\Sigma_0 \cup C)$-terms containing only constants common to A_0 and B_0 such that $\mathcal{H} = \mathcal{K}_0 \wedge \mathsf{Con}[D_A \wedge D_B]_0$ can be separated as described in Proposition 6, $A_0 \wedge B_0 \wedge (\mathcal{H} \backslash \mathcal{H}_{\mathsf{mix}}) \wedge \mathcal{H}_{\mathsf{sep}} \wedge \neg aRb$ is logically equivalent w.r.t. $\mathcal{T}_0$ with a separated conjunction of ground literals $\overline{A}_0 \wedge \overline{B}_0 \wedge \neg aRb$, which is therefore unsatisfiable, so $\overline{A}_0 \wedge \overline{B}_0 \models aRb$. From the P-interpolation property in $\mathcal{T}_0$, there exists a term containing the shared constants such that $\overline{A}_0 \wedge \overline{B}_0 \models_{\mathcal{T}_0} aRt \ \wedge \ tRb$. If we now replace all constants $c_{f(t_1,\dots,t_n)}$ introduced in the purification process or in the separation process with the terms they denote, we obtain $A \wedge B \models_{\mathcal{T}} aRt \wedge tRb$. □

We obtain the following procedure for P-interpolation if $A \wedge B \models_{\mathcal{T}} aRb$:

Step 1: Preprocess Using locality, flattening and purification we obtain a set $\mathcal{H} \wedge A_0 \wedge B_0$ of formulae in the base theory, where $\mathcal{H}$ is as in Proposition 6.

Step 2: $\Delta := \mathsf{T}$. Repeat as long as $A_0 \wedge B_0 \wedge \Delta \not\models aRb$:
Let $C \in \mathcal{H}$ whose premise is entailed by $A_0 \wedge B_0 \wedge \Delta$.
If C is not mixed, move C to $\mathcal{H}_{\mathsf{sep}}$ and add its conclusion to Δ.
If C is mixed, compute terms t_i which separate the premises in C, and separate the clause into an instance C_1 of monotonicity and an instance C_2 of a clause in $\mathcal{K}$ as in the proof of Proposition 6. Remove C from $\mathcal{H}$, and add C_1, C_2 to $\mathcal{H}_{\mathsf{sep}}$ and their conclusions to Δ.

Step 3: Compute separating term. Compute a separating term for $A_0 \wedge B_0 \wedge \Delta \models aRb$ in $\mathcal{T}_0$, and construct an interpolant for the extension as explained in the proof of Theorem 7.

5 Example: Semilattices with Monotone Operators

We will now analyze $\leq$-interpolation properties for theories of semilattices with monotone operators. A semilattice $(S, \sqcap)$ is set S with a binary operation $\sqcap$ which is associative, commutative and idempotent. One can equivalently regard semilattices as partially ordered sets $(S, \leq)$, in which infima of finite non-empty subsets exist; then $a \leq b$ iff $a \sqcap b = a$.

The theory SLat of semilattices can be axiomatized by equations (associativity, commutativity and idempotence of $\sqcap$) hence clearly is $\approx$-convex: Convexity w.r.t. $\leq$ follows from the fact that $x \leq y$ iff $(x \sqcap y) \approx x$. The theory SLat is $\leq$-interpolating, therefore also $\approx$-interpolating (cf. also [17]; we present the idea of the proof since it indicates how the intermediate terms can be computed):

Lemma 8. *The theory* SLat *of semilattices is $\leq$-interpolating.*

Proof (Idea): This is a constructive proof based on the fact that every semilattice is isomorphic to a sublattice of a power of S_2, where S_2 is the 2-element semilattice (or, alternatively, that every semilattice is isomorphic to a semilattice of sets). We prove that if A and B are two conjunctions of literals and $A \wedge B \models_{\mathsf{SLat}} a \leq b$, where a is a constant occurring in A and b a constant occurring in B, then there exists a term containing only common constants in A and B such that $A \wedge B \models_{\mathsf{SLat}} a \leq t$ and $A \wedge B \models_{\mathsf{SLat}} t \leq b$. We can assume without loss of generality that A and B consist only of atoms (for details cf. [17]). $A \wedge B \models_{\mathsf{SLat}} a \leq b$ if and only if the following conjunction of literals in propositional logic is unsatisfiable:

$$N_A\colon \left\{ \begin{array}{rl} P_{e_1 \sqcap e_2} \leftrightarrow P_{e_1} \wedge P_{e_2} & \\ P_{e_1} \leftrightarrow P_{e_2} & e_1 \approx e_2 \in A \\ P_{e_1} \rightarrow P_{e_2} & e_1 \leq e_2 \in A \\ \text{for all } e_1, e_2 \text{ subterms in } A & \end{array} \right. \quad N_B\colon \left\{ \begin{array}{rl} P_{g_1 \sqcap g_2} \leftrightarrow P_{g_1} \wedge P_{g_2} & \\ P_{g_1} \leftrightarrow P_{g_2} & g_1 \approx g_2 \in B \\ P_{g_1} \rightarrow P_{g_2} & g_1 \leq g_2 \in B \\ \text{for all } g_1, g_2 \text{ subterms in } B & \end{array} \right.$$

$$P_a \qquad\qquad\qquad\qquad \neg P_b$$

We obtain an unsatisfiable set of clauses $(N_A \wedge P_a) \wedge (N_B \wedge \neg P_b) \models \perp$, where N_A and N_B are sets of Horn clauses in which each clause contains a positive literal. We show that if $A \wedge B \models_{\mathsf{SLat}} a \leq b$ holds, then for the term

$$t := \bigsqcap \{ e \mid A \models_{\mathsf{SLat}} a \leq e, e \text{ common subterm of } A \text{ and } B \}$$

we have (i) $A \models_{\mathsf{SLat}} a \leq t$, and (ii) $A \wedge B \models_{\mathsf{SLat}} t \leq b$.

Clearly, $A \models_{\mathsf{SLat}} a \leq t$, thus (i) holds. For proving (ii), we analyze the set of clauses obtained by saturating $N_A \wedge P_a$ under ordered resolution in which all propositional variables occurring in A but not in B are larger than the common symbols. It is proved that for deriving the contradiction only the unit clauses P_e, where e is a common subterm of A and B and $A \models a \leq e$, and certain resolvents of $N_A \wedge P_a$ are needed. The full proof is given in [17] and also in [18]. □

We illustrate the computation of intermediate terms on an example.

Example 3. Let $A = \{a_1 \leq c_1,\ c_2 \leq a_2,\ a_2 \leq c_3\}$ and $B = \{c_1 \leq b_1,\ b_1 \leq c_2,\ c_3 \leq b_2\}$. It is easy to see that $A \wedge B \models a_1 \leq b_2$. We can find an intermediate term by using the methods described in the proof of Lemma 8: We saturate the set of clauses

$N_A \wedge P_{a_1} = (P_{a_1} \rightarrow P_{c_1}) \wedge (P_{c_2} \rightarrow P_{a_2}) \wedge (P_{a_2} \rightarrow P_{c_3}) \wedge P_{a_1}$

under ordered resolution, in which the propositional variables P_{a_1}, P_{a_2} are larger than $P_{c_1}, P_{c_2}, P_{c_3}$. This yields the clauses P_{c_1} and $P_{c_2} \rightarrow P_{c_3}$ containing shared propositional variables. $(N_A \wedge P_{a_1}) \wedge (N_B \wedge \neg P_{b_2})$ is unsatisfiable iff $N_B \wedge \neg P_{b_2} \wedge P_{c_1} \wedge (P_{c_2} \rightarrow P_{c_3})$ is unsatisfiable. Indeed $t = c_1$ is an intermediate term, as $A \models a_1 \leq c_1$ and $A \wedge B \models c_1 \leq b_2$. Note that $N_B \wedge \neg P_{b_2} \wedge P_{c_1}$ is satisfiable, so $B \not\models c_1 \leq b_2$. Moreover, we only need $P_{c_2} \rightarrow P_{c_3}$ in addition to $N_B \cup \neg P_{b_2}$ to derive $\perp$, thus $A \wedge B \models c_1 \leq b_2$ and the clause $P_{c_2} \rightarrow P_{c_3}$ obtained from N_A is really needed for this. □

Semilattices with operators. Let Σ be a set of unary[2] function symbols. We consider the extension $\mathsf{SLat}_\Sigma = \mathsf{SLat} \cup \mathsf{Mon}(\Sigma)$ of SLat with new function symbols in Σ satisfying the monotonicity axioms $\mathsf{Mon}_\Sigma = \bigcup_{f \in \Sigma} \mathsf{Mon}(f)$, where:

$\mathsf{Mon}(f) \qquad \forall x, y (x \leq y \rightarrow f(x) \leq f(y))$

and also extensions $\mathsf{SLat} \cup \mathsf{Mon}(\Sigma) \cup \mathcal{K}$, where $\mathcal{K}$ is a set of axioms of the form:

$$\forall x \quad f(x) \leq g(x) \tag{2}$$
$$\forall x, y \quad y \leq g(x) \rightarrow f(y) \leq h(x) \tag{3}$$

where $f, g, h \in \Sigma$, not necessarily all different.

Lemma 9. *The following extensions satisfy a locality property:*

(i) *The theory of semilattices* SLat *is local.*
(ii) $\mathsf{SLat} \cup \mathsf{Mon}_\Sigma$ *is a local extension of* SLat.
(iii) $\mathsf{SLat} \cup \mathsf{Mon}_\Sigma \cup \mathcal{K}$ *is a* Ψ*-local extension of* SLat*, where* Ψ *is the closure operator on ground terms defined as follows:*

$$\Psi(G) = \bigcup_{i \geq 0} \Psi^i(G), \text{ with } \Psi^0(G) = \mathsf{est}(G) \text{ (the set of ground terms in } G \text{ starting with extension functions), and}$$

$$\begin{aligned}\Psi^{i+1}(G) = {} & \{h(c) \mid \forall x (g(x) \leq h(x)) \in \mathcal{K} \text{ and } g(c) \in \Psi^i(G)\} \cup \\ & \{g(c) \mid \forall x (g(x) \leq h(x)) \in \mathcal{K} \text{ and } h(c) \in \Psi^i(G)\} \cup \\ & \{h(c) \mid \forall x, y (y \leq g(x) \rightarrow f(y) \leq h(x)) \in \mathcal{K} \text{ and } g(c) \in \Psi^i(G)\} \cup \\ & \{g(c) \mid \forall x, y (y \leq g(x) \rightarrow f(y) \leq h(x)) \in \mathcal{K} \text{ and } h(c) \in \Psi^i(G)\}.\end{aligned}$$

Proof: (i) follows from a result on the locality of lattices by Skolem [20], or by results in [9], since every partial semilattice weakly embeds into a total one. (ii) follows from results in [27,28]. (iii) Since the axioms in $\mathcal{K}$ are not always

[2] We assume that the function symbols are unary to simplify the presentation, and because in the applications to description logics we need only unary function symbols. All the results can be extended to function symbols of higher arity.

linear, we use the locality criterion for non-linear sets of clauses mentioned in Theorem 5, and the fact that every semilattice $P = (S, \sqcap, \{f\}_{f \in \Sigma})$ with partially defined monotone operators satisfying the axioms $\mathcal{K}$, and with the property that if a variable occurs in two terms $g(x), h(x)$ in a clause in $\mathcal{K}$, then for every $s \in S$, $g(s)$ is defined iff $h(s)$ is defined, weakly embeds into a semilattice with totally defined operators satisfying $\mathcal{K}$, which was proved in Lemma 4.5 from [26]. □

Given two sets of conjunctions of ground literals A and B over the signature of semilattices with operators, we consider the lattice operation $\sqcap$ to be interpreted and the function symbols in Σ to be uninterpreted. Let Σ_A be the function symbols in Σ occurring in A and Σ_B those occurring in B. We consider the following variants for "shared uninterpreted function symbols":

- *Intersection-sharing:* The shared function symbols of A and B are the function symbols in $\Sigma_A \cap \Sigma_B$.
- *$\Theta_{\mathcal{K}}$-sharing:* Let $\Theta_{\mathcal{K}}(\Sigma_A)$ and $\Theta_{\mathcal{K}}(\Sigma_B)$ be defined as explained in Example 1. The $\Theta_{\mathcal{K}}$-shared function symbols are the function symbols in $\Theta_{\mathcal{K}}(\Sigma_A) \cap \Theta_{\mathcal{K}}(\Sigma_B)$.

Theorem 10. *For every set $\mathcal{K}$ containing clauses of the form (2) and (3) above, the theory $\mathsf{SLat} \cup \mathsf{Mon}_\Sigma \cup \mathcal{K}$ of semilattices with monotone operators satisfying axioms $\mathcal{K}$ is $\leq$-interpolating with the notion of $\Theta_{\mathcal{K}}$-sharing for uninterpreted function symbols.*

Proof: The clauses of type (2) and (3) satisfy the conditions in the statement of Proposition 6 and Theorem 7. The result is therefore a consequence of the fact that SLat is convex and $\{\approx, \leq\}$-interpolating, and of Proposition 6 and Theorem 7. □

We illustrate the way Theorem 4, Proposition 6 and Theorem 7 and the algorithm in Sect. 4 can be used for computing intermediate terms below:

Example 4. Consider the extension $\mathsf{SLO} = \mathsf{SLat} \cup \mathsf{Mon}_f \cup \mathsf{Mon}_g \cup \mathcal{K}$ of SLat with two monotone functions f, g satisfying: $\mathcal{K} = \{y \leq g(x) \rightarrow f(y) \leq g(x)\}$. Consider the following conjunctions of atoms: $A := d \leq g(a) \wedge a \leq c \wedge g(c) \leq a$ and $B := b \leq d \wedge b \leq f(b)$. It can be checked that $A \wedge B \models b \leq a$.

To obtain a separating term we proceed as follows: By the definition of SLO, $A \wedge B \models_{SLO} b \leq a$ iff $\mathsf{SLat} \wedge \mathsf{Mon}_f \wedge \mathsf{Mon}_g \wedge \mathcal{K} \wedge A \wedge B \wedge \neg(b \leq a) \models \perp$. By Theorem 4, this is the case iff $\mathsf{SLat} \wedge (\mathsf{Mon}_f \wedge \mathsf{Mon}_g \wedge \mathcal{K})[\Psi(G)] \wedge G \models \perp$, where $G = A \wedge B \wedge \neg(b \leq a)$, $\mathsf{est}(G) = \{g(a), g(c), f(b)\}$ and $\Psi(G) = \{g(a), g(c), f(b)\}$.

- $\mathsf{Mon}_f[\Psi(G)] = \{b \leq b \rightarrow f(b) \leq f(b)\}$ (redundant).
- $\mathsf{Mon}_g[\Psi(G)] = \{d_1 \leq d_2 \rightarrow g(d_1) \leq g(d_2) \mid d_1, d_2 \in \{a, c\}\}$.
- $\mathcal{K}[\Psi(G)] = \{b \leq g(a) \rightarrow f(b) \leq g(a), b \leq g(c) \rightarrow f(b) \leq g(c)\}$.

Step 1: We purify $(\mathsf{Mon}_f \wedge \mathsf{Mon}_g \wedge \mathcal{K})[\Psi(G)] \wedge G$, by introducing constants a_1 for $g(a)$, c_1 for $g(c)$ and b_1 for $f(b)$ and obtain the formula $\mathsf{Def} \wedge A_0 \wedge B_0 \wedge \mathsf{Mon}_0 \wedge \mathcal{K}_0$:

Def	$A_0 \wedge B_0$	$\mathsf{Mon}_0 \wedge \mathcal{K}_0$	
$D_A : a_1 \approx g(a) \wedge c_1 \approx g(c)$	$A_0 : d \leq a_1 \wedge a \leq c \wedge c_1 \leq a$	Mon_A	$a \lhd c \rightarrow a_1 \lhd c_1$
$D_B : b_1 \approx f(b)$	$B_0 : b \leq d \ \wedge b \leq b_1$	$\mathcal{K}_{\mathsf{mix}}$	$b \leq a_1 \rightarrow b_1 \leq a_1$
	$\lhd \in \{\leq, \geq\}$		$b \leq c_1 \rightarrow b_1 \leq c_1$

Step 2. $\Delta := \top$. Find clauses in $\mathsf{Mon}_0 \wedge \mathcal{K}_0$ with premises entailed by $A_0 \wedge B_0 \wedge \Delta$.

$C = a \leq c \rightarrow a_1 \leq c_1$: C is not mixed. Since $A_0 \wedge B_0 \models_{\mathsf{SLat}} a \leq c$, $A_0 \wedge B_0 \wedge (a \leq c \rightarrow a_1 \leq c_1)$ is equivalent to $A_0 \wedge B_0 \wedge a_1 \leq c_1$. Let $\Delta := \{a_1 \leq c_1\}$.

$C = b \leq a_1 \rightarrow b_1 \leq a_1$: C is mixed. Since $A_0 \wedge B_0 \wedge a_1 \leq c_1 \models b \leq a_1$ we find a separating term. For this we use the method described in the proof of Lemma 8. We consider the encoding $N_B \wedge P_b := (P_b \rightarrow P_d) \wedge (P_b \rightarrow P_{b_1}) \wedge P_b$. Using ordered resolution with an ordering in which $P_b, P_{b_1} \succ P_d$ we derive the unit clauses P_d and P_{b_1}. Since d is the only shared constant, $t = d$ is the separating term. Thus, $A_0 \wedge B_0 \wedge a_1 \leq c_1 \models b \leq d \ \wedge \ d \leq a_1$. We now can separate the instance $b \leq a_1 \rightarrow b_1 \leq a_1$ of the clause in $\mathcal{K}$ by introducing a new shared constant d_1 as a name for $f(d)$ and replacing the clause, as described in the algorithm at the end of Sect. 4, with the conjunction of

(1) $b \leq d \rightarrow b_1 \leq d_1$ and
(2) $d \leq a_1 \rightarrow d_1 \leq a_1$

((1) is an instance of a monotonicity axiom, (2) is another instance of $\mathcal{K}$), and $A_0 \wedge B_0 \wedge a_1 \leq c_1 \wedge (b \leq d \rightarrow b_1 \leq d_1) \wedge (d \leq a_1 \rightarrow d_1 \leq a_1)$ is equivalent to $A_0 \wedge B_0 \wedge a_1 \leq c_1 \wedge b_1 \leq d_1 \wedge d_1 \leq a_1$. Let $\Delta := \Delta \wedge b_1 \leq d_1 \wedge d_1 \leq a_1$.

Step 3: The last conjunction entails $b \leq a$. To compute a separating term, we again use Lemma 8. We consider the encoding $N'_B \wedge P_b := (P_b \rightarrow P_d) \wedge (P_b \rightarrow P_{b_1}) \wedge (P_{b_1} \rightarrow P_{d_1}) \wedge P_b$ of the B-part of the conjunction, $B_0 \wedge b_1 \leq d_1$. Using ordered resolution with an ordering in which $P_b, P_{b_1} \succ P_d, P_{d_1}$ we derive the unit clauses P_d, P_{b_1} and P_{d_1}. Since d, d_1 are the shared constants, $t = d \sqcap d_1$ is the separating term. (It can be seen that already d is a separating term.) □

If $\mathcal{K}$ contains axioms of type (3) then the theory of semilattices with operators is not $\leq$-interpolating when sharing is regarded as *intersection-sharing.* Indeed, assume that for every $\mathcal{K}$ containing axioms of type (3), $\mathsf{SLat}_\Sigma(\mathcal{K})$ is $\leq$-interpolating w.r.t. intersection-sharing. Then it would also be $\approx$-interpolating w.r.t. intersection-sharing. This cannot be the case, as can be seen from the following example.

Example 5. Consider the theory $\mathsf{SLat}_\Sigma(\mathcal{K})$ of semilattices with monotone operators f, g satisfying the axioms $\mathcal{K} = \{x \leq g(y) \rightarrow f(x) \leq g(y)\}$, and let C be a set of constants containing constants a, b, d, e. We show that this theory does not have the Σ_S-Beth-definability property, where $\Sigma_S = \{g, e\}$.

Consider the conjunction of literals $A = (a \leq f(e)) \wedge (e \leq g(b)) \wedge (g(b) \leq a)$. One can prove that a is implicitly definable w.r.t. $\{g, e\}$ by proving, using the hierarchical reduction for local theory extensions in Theorem 4, that:

$$(a \leq f(e)) \wedge (e \leq g(b)) \wedge (g(b) \leq a) \wedge (a' \leq f'(e)) \wedge (e \leq g(b')) \wedge (g(b') \leq a') \models_{\mathsf{Slat}_\Sigma(\mathcal{K} \cup \mathcal{K}')} a \approx a'.$$

We show that a is not explicitly definable w.r.t. $\{g, e\}$. If there exists a term t containing only g and e such that $(a {\leq} f(e)) \wedge (e {\leq} g(b)) \wedge (g(b) {\leq} a) \models_{\mathsf{Slat}_{\Sigma}(\mathcal{K})} a {\approx} t$, then the interpretations of a and t are equal in every model of $\mathsf{SLat}_{\Sigma}(\mathcal{K})$ which is a model of A. We show that this is not the case. Let $S = (\{a, e, b, d\}, \sqcap, f, g)$ be the semilattice where $d \leq e \leq a$, $d \leq b$ and $a {\sqcap} b = e {\sqcap} b = d$, and $f(a) = f(e) = a$, $f(b) = f(d) = d$, $g(a) = g(e) = g(d) = d$ and $g(b) = a$. Then S satisfies A, f and g are monotone, and S is a model of $\mathcal{K}$: Assume that $x \leq g(y)$. If $y \in \{a, e, d\}$ then $g(y) = d$ so $x = d$, and $f(d) = d \leq g(y)$. If $y = b$ then $g(b) = a$, so x can be a, e or d, and $f(a) = f(e) = a$, $f(d) = d$, so $f(x) \leq g(b) = a$. A term t containing only g and e can be e or can contain occurrences of g. If $t = e$ then the interpretation of t in S is not a. If t contains occurrences of g it can be proven that the interpretation of t in S is d, i.e. is again different from a.
Thus $\mathcal{T} = \mathsf{SLat}_{\Sigma}(\mathcal{K})$ does not have the Beth definability property w.r.t. Σ_S, hence, by Theorem 3, $\mathcal{T} \cup \mathcal{T}' = \mathsf{SLat}_{f,g}(\mathcal{K}) \cup \mathsf{SLat}_{f',g}(\mathcal{K}') = \mathsf{SLat}_{f,f',g}(\mathcal{K} \cup \mathcal{K}')$, where $\mathcal{K}' = \{y \leq g(x) \rightarrow f'(y) \leq g(x)\}$, does not have the $\approx$-interpolation property w.r.t. intersection-sharing, hence it does not have the $\leq$-interpolation property w.r.t. intersection-sharing. (By Theorem 10 and Theorem 3, $\mathcal{T}$ has the $\Theta_{\mathcal{K}}(\Sigma_S)$-Beth definability property, where $\Theta_{\mathcal{K}}(\Sigma_S) = \{f, g, e\}$. Indeed, then $A \models a \approx f(e)$.) □

6 Applications to $\mathcal{EL}$ and $\mathcal{EL}^+$-Subsumption

We now explain how these results can be used in the study of the description logics $\mathcal{EL}$ and $\mathcal{EL}^+$. In any description logic a set N_C of *concept names* and a set N_R of *roles* is assumed to be given. *Concept descriptions* can be defined with the help of a set of *concept constructors*. The available constructors determine the expressive power of a description logic. If we only allow intersection and existential restriction as concept constructors, we obtain the description logic $\mathcal{EL}$ [1], a logic used in terminological reasoning in medicine [29,30]. The table below shows the constructor names used in $\mathcal{EL}$ and their semantics.

Constructor name	Syntax	Semantics
conjunction	$C_1 \sqcap C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
existential restriction	$\exists r.C$	$\{x \mid \exists y((x, y) \in r^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}})\}$

The semantics is given by interpretations $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$, where $C^{\mathcal{I}} \subseteq \Delta$ and $r^{\mathcal{I}} \subseteq \Delta^2$ for every $C \in N_C$, $r \in N_R$. The extension of $\cdot^{\mathcal{I}}$ to concept descriptions is inductively defined using the semantics of the constructors. In [2,3], the extension $\mathcal{EL}^+$ of $\mathcal{EL}$ with role inclusion axioms is studied.

A TBox (or terminology) is a finite set consisting of *general concept inclusions* (GCI) of the form $C \sqsubseteq D$, where C and D are concept descriptions. A CBox consists of a TBox and a set of role inclusions of the form $r_1 \circ \cdots \circ r_n \sqsubseteq s$, so we view CBoxes as unions $GCI {\cup} \mathcal{R}$ of a set GCI of general concept inclusions and a set $\mathcal{R}$ of role inclusions of the form $r_1 \circ \cdots \circ r_n \sqsubseteq s$, with $n \geq 1$.[3] An interpretation $\mathcal{I}$ is a *model of the CBox* $\mathcal{C} = GCI \cup \mathcal{R}$ if it is a model of GCI, i.e., $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every $C \sqsubseteq D \in GCI$, and satisfies all role inclusions in $\mathcal{C}$, i.e., $r_1^{\mathcal{I}} \circ \cdots \circ r_n^{\mathcal{I}} \subseteq s^{\mathcal{I}}$ for all $r_1 \circ \cdots \circ r_n \subseteq s \in \mathcal{R}$. If $\mathcal{C}$ is a CBox and C_1, C_2 are concept descriptions, then $\mathcal{C} \models C_1 \sqsubseteq C_2$ if and only if $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{C}$.

In [23] we studied the link between TBox subsumption in $\mathcal{EL}$ and uniform word problems in the corresponding classes of semilattices with monotone functions. In [25], we showed that these results naturally extend to CBoxes and to the description logic $\mathcal{EL}^+$. When defining the semantics of $\mathcal{EL}$ or $\mathcal{EL}^+$ with role names N_R we use a class of $\sqcap$-semilattices with monotone operators of the form SLat_{Σ}, where $\Sigma = \{f_r \mid r \in N_R\}$. Every concept description C can be represented as a term $\overline{C}$; the encoding is inductively defined: Every concept name $C \in N_C$ is regarded as a constant $\overline{C} = C$. We define $\overline{C_1 \sqcap C_2} := \overline{C}_1 \sqcap \overline{C}_2$ and $\overline{\exists r C} = f_r(\overline{C})$. If $\mathcal{R}$ is a set of role inclusions of the form $r \sqsubseteq s$ and $r_1 \circ r_2 \sqsubseteq s$, let $\mathcal{K}$ be the set of all axioms of the form:

$$\begin{array}{ll} \forall x \; (f_r(x) \leq f_s(x)) & \text{for all } r \sqsubseteq s \in \mathcal{R} \\ \forall x \; (f_{r_1}(f_{r_2}(x)) \leq f_s(x)) & \text{for all } r_1 \circ r_2 \sqsubseteq s \in \mathcal{R} \end{array}$$

Theorem 11 ([25]). *Assume that the only concept constructors are intersection and existential restriction. Then for all concept descriptions D_1, D_2 and every $\mathcal{EL}^+$ CBox $\mathcal{C}{=}GCI{\cup}\mathcal{R}$ – where $\mathcal{R}$ consists of role inclusions of the form $r \sqsubseteq s$ and $r_1 \circ r_2 \sqsubseteq s$ – with concept names $N_C = \{C_1, \ldots, C_n\}$ and set of roles N_R:*

$$\mathcal{C} \models D_1 {\sqsubseteq} D_2 \quad \textit{iff} \quad \Big(\bigsqcap_{C \sqsubseteq D \in GCI} \overline{C} {\leq} \overline{D} \Big) \models_{\mathsf{SLat}_{\Sigma}(\mathcal{K})} \overline{D_1} {\leq} \overline{D_2},$$

where Σ is associated with N_R and $\mathcal{K}$ with $\mathcal{R}$ as described above.

In [8,31] the following notion of interpolation which we call $\sqsubseteq$-interpolation is defined: A description logic has the $\sqsubseteq$-interpolation property if for any CBoxes $\mathcal{C}_A = GCI_A \cup \mathcal{R}_A, \mathcal{C}_B = GCI_B \cup \mathcal{R}_B$ and any concept descriptions C, D such that $\mathcal{C}_A \cup \mathcal{C}_B \models C \sqsubseteq D$ there exists a concept description T containing only concept and role symbols "shared" by $\{\mathcal{C}_A, C\}$ and $\{\mathcal{C}_B, D\}$ such that $\mathcal{C}_A \cup \mathcal{C}_B \models C \sqsubseteq T$ and $\mathcal{C}_A \cup \mathcal{C}_B \models T \sqsubseteq D$. By Theorem 11, $\mathcal{C}_A \cup \mathcal{C}_B \models C \sqsubseteq D$ iff $A \wedge B \models_{\mathsf{SLat}_{\Sigma}(\mathcal{K})}$ $\overline{C} {\leq} \overline{D}$, where $A = \bigsqcap_{C_1 \sqsubseteq C_2 \in GCI_A} \overline{C_1} {\leq} \overline{C_2}$, $B = \bigsqcap_{C_1 \sqsubseteq C_2 \in GCI_B} \overline{C_1} {\leq} \overline{C_2}$, and $\mathcal{K} = \mathcal{K}_A \cup \mathcal{K}_B$, the union of the axioms associated with the set inclusions $\mathcal{R}_A$ resp. $\mathcal{R}_B$. By Theorem 10, there exists a term containing only constants and function symbols $\Theta_{\mathcal{K}_A \cup \mathcal{K}_B}$*-shared* by A and B such that $A \wedge B \models_{\mathsf{SLat}_{\Sigma}(\mathcal{K}_A \cup \mathcal{K}_B)} \overline{C} {\leq} t \wedge t {\leq} \overline{D}$. From t we can construct a concept description T containing only concept names and roles *shared* by $\mathcal{C}_A$ and $\mathcal{C}_B$, and by Theorem 11, $\mathcal{C}_A \wedge \mathcal{C}_B \models C \sqsubseteq T \wedge T \sqsubseteq D$. Therefore, the $\sqsubseteq$-interpolation problem studied for description logics in [8,31] can be expressed in the case of $\mathcal{EL}$ and $\mathcal{EL}^+$ as a $\leq$-interpolation problem in the class of semilattices with operators, and the hierarchical method for $\leq$-interpolation can be used in this case. We distinguish between intersection-sharing and $\Theta_{\mathcal{R}}$-sharing, where $\Theta_{\mathcal{R}}$ is the analogon of $\Theta_{\mathcal{K}}$ where $\mathcal{K}$ is the translation of $\mathcal{R}$.

[3] It can be shown that it is sufficient to consider role inclusions of the form $r \sqsubseteq s$ or $r_1 \circ r_2 \sqsubseteq s$, where r, s, r_1, r_2 are role names [3].

Corollary 12. *$\mathcal{EL}$ and $\mathcal{EL}^+$ have the $\sqsubseteq$-interpolation property w.r.t. $\Theta_{\mathcal{R}}$-sharing. $\mathcal{EL}^+$ with role inclusions of the form $r_1 \circ r_2 \sqsubseteq s$ does not have $\sqsubseteq$-interpolation w.r.t. intersection-sharing.*

7 Conclusions and Future Work

In this paper we gave a hierarchical method for P-interpolation in certain classes of local theory extensions $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$. We used these results for proving $\leq$-interpolation in classes of semilattices with monotone operators satisfying additional clauses $\mathcal{K}$ with a suitable notion of $\Theta_{\mathcal{K}}$-sharing we defined. We defined a form of Beth definability w.r.t. a subsignature Σ_S and used it to show that the class of semilattices with operators under consideration does not have the $\leq$-interpolation property if only the common function symbols and constants are considered to be "shared". We discussed how these results can be used for the study of interpolation in $\mathcal{EL}$ and $\mathcal{EL}^+$.

The ideas were implemented in a prototype implementation[4] for the theory of semilattices with operators satisfying axioms of type (1) considered in this paper. The program is written in Python and uses Z3 [7] and SPASS [33] as external provers. The program implements Steps 1–3 in the algorithm presented at the end of Sect. 4 with the following optimization: In Step 1 after instantiation and purification, in order to reduce the size of the set of instances of axioms to be considered, an unsatisfiable core is computed with Z3. The program separates the mixed instances by computing intermediate terms for their premises using Theorem 8 and Proposition 6; for applying ordered resolution the prover SPASS is used. In Step 3, the intermediate term T for $C \leq D$ is computed using the method described in Theorem 8, again using SPASS.

For the use for interpolation in $\mathcal{EL}$ and $\mathcal{EL}^+$, the CBoxes $\mathcal{C}_A$ and $\mathcal{C}_B$ and the subsumption $C \sqsubseteq D$ are given as an input. A minimal subset of $\mathcal{C}_A \cup \mathcal{C}_B$ is computed from which $C \sqsubseteq D$ can be derived. (The user can choose between a precise translation to SPASS or a propositional translation to Z3 which is not always precise, but turned out to be a good approximation. Standard implementations available for computing justifications of entailments from description logic ontologies could be used as well.) The problem is then translated into a problem for $\leq$-interpolation in semilattices with operators. After computing the interpolating term, the result is expressed in the syntax of description logics.

In future work we will explore other application areas of these results, both to classes of non-classical logics and to theories relevant in the verification. We plan to extend the implementation with possibilities of choosing the base theory and the methods for P-interpolation in the base theory. We will further investigate the links with Beth definability and possibilities of using Beth definability for computing explicit definitions for implicitly definable terms – and analyze the applicability of such results in description logics but also in verification.

[4] The implementation and some tests can be found here: https://userpages.uni-koblenz.de/~sofronie/p-interpolation-and-el/.

Acknowledgments. We thank the reviewers for their helpful comments. The research reported here was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Projektnummer 465447331.

References

1. Baader, F.: Terminological cycles in a description logic with existential restrictions. In: Gottlob, G., Walsh, T. (eds.) Proceedings of the 18th International Joint Conference on Artificial Intelligence, IJCAI 2003, pp. 325–330. Morgan Kaufmann (2003)
2. Baader, F., Lutz, C., Suntisrivaraporn, B.: Efficient reasoning in $\mathcal{EL}^+$. In: Parsia, B., Sattler, U., Toman, D. (eds.) Proceedings of the 2006 International Workshop on Description Logics (DL 2006), CEUR Workshop Proceedings, vol. 189. CEUR-WS.org (2006)
3. Baader, F., Lutz, C., Suntisrivaraporn, B.: Is tractable reasoning in extensions of the description logic $\mathcal{EL}$ useful in practice? J. Logic Lang. Inf. (2007). Special issue on Method for Modality (M4M)
4. Bruttomesso, R., Ghilardi, S., Ranise, S.: Quantifier-free interpolation in combinations of equality interpolating theories. ACM Trans. Comput. Log. **15**(1), 5:1–5:34 (2014)
5. Burris, S., Sankappanavar, H.P.: A Course in Universal Algebra. Graduate Texts in Mathematics, vol. 78. Springer, Heidelberg (1981)
6. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Combination of uniform interpolants via Beth definability. J. Autom. Reason. **66**(3), 409–435 (2022)
7. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
8. Fortin, M., Konev, B., Wolter, F.: Interpolants and explicit definitions in extensions of the description logic $\mathcal{EL}$. In: Kern-Isberner, G., Lakemeyer, G., Meyer, T. (eds.) Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning, KR 2022 (2022)
9. Ganzinger, H.: Relating semantic and proof-theoretic concepts for polynominal time decidability of uniform word problems. In: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, 16–19 June 2001, pp. 81–90. IEEE Computer Society (2001)
10. Hodges, W.: Model Theory. Encyclopedia of Mathematics and Its Applications, vol. 42. Cambridge University Press (1993)
11. Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: On local reasoning in verification. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 265–281. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_19
12. Ihlemann, C., Sofronie-Stokkermans, V.: On hierarchical reasoning in combinations of theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 30–45. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_4
13. Kovács, L., Voronkov, A.: Interpolation and symbol elimination. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 199–213. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_17

14. Krajícek, J.: Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. J. Symb. Log. **62**(2), 457–486 (1997)
15. McAllester, D.A.: Automatic recognition of tractability in inference relations. J. ACM **40**(2), 284–303 (1993)
16. McMillan, K.L.: An interpolating theorem prover. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 16–30. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_2
17. Peuter, D., Sofronie-Stokkermans, V.: Finding high-level explanations for subsumption w.r.t. combinations of CBoxes in $\mathcal{EL}$ and $\mathcal{EL}^+$. In: Borgwardt, S., Meyer, T. (eds.) Proceedings of the 33rd International Workshop on Description Logics (DL 2020) co-located with the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR 2020), CEUR Workshop Proceedings, vol. 2663. CEUR-WS.org (2020)
18. Peuter, D., Sofronie-Stokkermans, V., Thunert, S.: On P-interpolation in local theory extensions and applications to the study of interpolation in the description logics $\mathcal{EL}, \mathcal{EL}^+$. (Extended version) CoRR, https://arxiv.org/abs/2307.08843 (2023)
19. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint solving for interpolation. J. Symb. Comput. **45**(11), 1212–1233 (2010)
20. Skolem, T.: Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theorem über dichte Mengen. In: Selected Works in Logic. Universitetsforlaget (1920)
21. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 219–234. Springer, Heidelberg (2005). https://doi.org/10.1007/11532231_16
22. Sofronie-Stokkermans, V.: Interpolation in local theory extensions. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 235–250. Springer, Heidelberg (2006). https://doi.org/10.1007/11814771_21
23. Sofronie-Stokkermans, V.: Automated theorem proving by resolution in non-classical logics. Ann. Math. Artif. Intell. **49**(1–4), 221–252 (2007)
24. Sofronie-Stokkermans, V.: Interpolation in local theory extensions. Log. Methods Comput. Sci. **4**(4) (2008)
25. Sofronie-Stokkermans, V.: Locality and subsumption testing in $\mathcal{EL}$ and some of its extensions. In: Areces, C., Goldblatt, R. (eds.) Advances in Modal Logic, vol. 7, pp. 315–339. College Publications (2008)
26. Sofronie-Stokkermans, V.: Representation theorems and locality for subsumption testing and interpolation in the description logics $\mathcal{EL}$, $\mathcal{EL}^+$ and their extensions with n-ary roles and numerical domains. Fundam. Inform. **156**(3–4), 361–411 (2017)
27. Sofronie-Stokkermans, V., Ihlemann, C.: Automated reasoning in some local extensions of ordered structures. In: 37th International Symposium on Multiple-Valued Logic, ISMVL 2007, p. 1. IEEE Computer Society (2007)
28. Sofronie-Stokkermans, V., Ihlemann, C.: Automated reasoning in some local extensions of ordered structures. Multiple-Valued Logic Soft Comput. **13**(4–6), 397–414 (2007)
29. Spackman, K.A.: Normal forms for description logic expressions of clinical concepts in SNOMED RT. In: American Medical Informatics Association Annual Symposium, AMIA 2001. AMIA (2001)
30. Spackman, K.A., Campbell, K.E., Côté, R.A.: SNOMED RT: a reference terminology for health care. In: American Medical Informatics Association Annual Symposium, AMIA 1997. AMIA (1997)

31. ten Cate, B., Franconi, E., Seylan, I.: Beth definability in expressive description logics. J. Artif. Intell. Res. **48**, 347–414 (2013)
32. Tinelli, C.: Cooperation of background reasoners in theory reasoning by residue sharing. J. Autom. Reason. **30**(1), 1–31 (2003)
33. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 140–145. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_10
34. Yorsh, G., Musuvathi, M.: A combination method for generating interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 353–368. Springer, Heidelberg (2005). https://doi.org/10.1007/11532231_26

Theorem Proving in Dependently-Typed Higher-Order Logic

Colin Rothgang[1(✉)], Florian Rabe[2], and Christoph Benzmüller[1,3]

[1] Mathematics and Computer Science, FU, Berlin, Germany
colin.rothgang@gmx.de
[2] Computer Science, University Erlangen-Nürnberg, Erlangen, Germany
[3] AI Systems Engineering, University Bamberg, Bamberg, Germany

Abstract. Higher-order logic HOL offers a very simple syntax and semantics for representing and reasoning about typed data structures. But its type system lacks advanced features where types may depend on terms. Dependent type theory offers such a rich type system, but has rather substantial differences to HOL, as well as comparatively poor proof automation support.

We introduce a dependently-typed extension DHOL of HOL that retains the style and conceptual framework of HOL. Moreover, we build a translation from DHOL to HOL and implement it as a preprocessor to a HOL theorem prover, thereby obtaining a theorem prover for DHOL.

1 Introduction and Related Work

Theorem proving in higher-order logic (HOL) [5,11] has been a long-running research strand producing multiple mature interactive provers [10,13,17] and automated provers [2,4,23]. Similarly, many, mostly interactive, theorem provers are available for various versions of dependent type theory (DTT) [7,9,15,18]. However, it is (maybe surprisingly) difficult to develop theorem provers for dependently-typed higher-order logic (DHOL).

In this paper, we use HOL to refer to a version of Church's *simply*-typed λ-calculus with a base type bool for Booleans, simple function types $\rightarrow$, and equality $=_A : A \rightarrow A \rightarrow \mathsf{bool}$. This already suffices to define the usual logical quantifiers and connectives.[1] Intuitively, it is straightforward to develop DHOL accordingly on top of the *dependently*-typed λ-calculus, which uses a dependent function type $\Pi x{:}A.\, B$ instead of $\rightarrow$. However, several subtleties arise that seem deceptively minor at first but end up presenting fundamental theoretical issues. They come up already in the elementary expression $x =_A y \Rightarrow f(x) =_{B(x)} f(y)$ for some dependent function $f : \Pi x{:}A.\, B(x)$.

Firstly, the equality $f(x) =_{B(x)} f(y)$ is not even well-typed because the terms $f(x) : B(x)$ and $f(y) : B(y)$ do not have the same type. Intuitively, it is obvious that the type system can (and maybe should) be adjusted so that the equality $x =_A y$ between terms

[1] We do not assume a choice operator or the axiom of infinity.

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 438–455, 2023.
https://doi.org/10.1007/978-3-031-38499-8_25

carries over to an equality $B(x) \equiv B(y)$ between types.[2] However, this means that the undecidability of equality leaks into the equality of types and thus into type-checking.

While some interactive provers successfully use undecidable type systems [6,16], most formal systems for DTT commit to keeping type-checking decidable. The typical approach goes back to Martin-Löf type theory [14] and the calculus of constructions [8] and uses two separate equality relations, a decidable meta-level equality for use in the type-checker and a stronger undecidable one subject to theorem proving. Moreover, it favors the propositions-as-types representation and deemphasizes or omits a type of classical Booleans. This approach has been studied extensively [7,9,15] and is not the subject of this paper.

Instead, our motivation is to retain a single equality relation and classical Booleans. This is arguably more intuitive to users, especially to those outside the DTT community such as typical HOL users or mathematicians, and it is certainly much closer to the logics of the strongest available ATP systems. This means we have to pay the price of undecidable type-checking. The current paper was prompted by the observation that this price may be acceptable for two reasons:

1. If our ultimate interest is theorem proving, undecidability comes up anyway. Indeed, it is plausible that the cost of showing the well-typedness of a conjecture will be negligible compared to the cost of proving it.

2. As the strength of ATPs for HOL increases, the practical drawbacks of undecidable type-checking decrease, which indicates revisiting the trade-off from time to time. Indeed, if we position DHOL close to an existing HOL ATP, it is plausible that the price will, in practice, be affordable.

Secondly, even if we add a rule like "if $\vdash x =_A y$, then $\vdash B(x) \equiv B(y)$" to our type system, the above expression is still not well-typed: Above, the equality $x =_A y$ on the left of $\Rightarrow$ is needed to show the well-typedness of the equality $f(x) =_{B(x)} f(y)$ on the right. This intertwines theorem proving and type-checking even further. Concretely, we need a *dependent implication*, where the first argument is assumed to hold while checking the well-typedness of the second one. Formally, this means that to show $\vdash F \Rightarrow G : \mathsf{bool}$, we require $\vdash F : \mathsf{bool}$ and $F \vdash G : \mathsf{bool}$. Similarly, we need a dependent conjunction. And if we are classical, we may also opt to add a dependent disjunction $F \vee G$, where $\neg F$ is assumed in G. Naturally, dependent conjunction and disjunction are not commutative anymore. This may feel disruptive, but similar behavior of connectives is well-known from short-circuit evaluation in programming languages.

The meta-logical properties of dependent connectives are straightforward. However, interestingly, these connectives can no longer be defined from just equality. At least one of them (we will choose dependent implication) must be taken as an additional primitive in DHOL along with $=_A$.

Finally, the above generalizations require a notion of DHOL-contexts that is more complex than for HOL. HOL-contexts can be stratified into (a) a set of variable declarations

[2] Note that while term equality $=_A$ is a bool-valued connective, type equality $\equiv$ is not. Instead, in HOL, $\equiv$ is a judgment at the same level as the typing judgment $t : A$.

$x_i : A_i$, and (b) a set of logical assumptions F possibly using the variables x_i. Moreover, the former are often not explicitly listed at all and instead inferred from the remainder of the sequent. But in DHOL, the well-formedness of an A_i may now depend on previous logical assumptions. To linearize this inter-dependency, DHOL contexts must consist of a single list alternating between variable declarations and assumptions.

Contribution. Our contribution is twofold. Firstly, we introduce a new logic DHOL designed along the lines described above. Moreover, we further extend DHOL with predicate subtypes $A|_p$ for a predicate $p : A \to \mathtt{bool}$ on the type A. Besides dependent types, these constitute a second important source of terms occurring in types. Because they also make typing undecidable, they are often avoided. The most prominent exception is PVS [16], whose kernel essentially arises by adding predicate subtypes to HOL. In current HOL ITPs going back to [10], their use is usually restricted to the subtype definition principle: here a definition $b := A|_p$ may occur on toplevel and is elaborated into a fresh type b that is axiomatized to mimic the subtype $A|_p$. Because we are committed to undecidable typing anyway, predicate subtypes fit naturally into our approach.

Secondly, we develop and implement a sound and complete translation of DHOL into HOL. This setup allows the use of DHOL as the expressive user-facing language and HOL as the internal theorem-proving language. We position our implementation close to an existing HOL ATP, namely the LEO-III system. From the LEO-III perspective, DHOL serves as an additional input language that is translated into HOL by an external logic embedding tool [21,22] in the LEO-III ecosystem. Because LEO-III already supports such embeddings and because the TPTP syntax [24] foresees the use of dependent types in ATPs and provides syntax for them (albeit without a normative semantics), we were able to implement the translation with no disruptions to existing workflows.

The general idea of our translation of dependent into simple type theory is not new [3]. In that work, Martin-Löf-style dependent type theory is translated into Gordon's HOL ITP [10]. This work differs critically from ours because it uses DTT in propositions-as-types style. Our work builds DHOL with classical Booleans and equality predicate, which makes the task of proving the translation sound and complete very different. Moreover, their work targets an interactive prover while ours targets automated ones.

Overview. In Sect. 2 we recap the HOL logic. In Sect. 3 we extend it to DHOL and define our translation from DHOL to HOL. In Sect. 4 we add subtyping and predicate subtypes. In Sect. 5 we prove the soundness and completeness of the translation. In Sect. 6 we describe how to use our translation and a HOL ATP to implement a theorem prover for DHOL.

2 Preliminaries: Higher-Order Logic

We introduce the syntax and rules of HOL. Our definitions are standard except that we tweak a few details in order to later present the extension to DHOL more succinctly. We use the following grammar for HOL:

$$\begin{array}{lcll}
T & ::= & \circ \mid T, a : \mathsf{tp} \mid T, c : A \mid T, c \colon F & \text{theories} \\
\Gamma & ::= & . \mid \Gamma, x : A \mid \Gamma, x \colon F & \text{contexts} \\
A, B & ::= & a \mid A \to B \mid \mathsf{bool} & \text{types} \\
s, t, f, F, G & ::= & c \mid x \mid \lambda x{:}A.\, t \mid f\ t \mid s =_A t \mid F \Rightarrow G & \text{terms}
\end{array}$$

A theory T is a list of base type declarations $a : \mathsf{tp}$, typed constant declarations $c : A$, and named axioms $c\colon F$ asserting the formula F. A context Γ has the same form except that no type variables are allowed. It is not strictly necessary to use named axioms and assumptions, but it makes our extensions to DHOL later on simpler. We write $\circ$ and . for the empty theory and context, respectively. At this point, it is possible to normalize contexts into a set of variable declarations followed by a set of assumptions because the well-formedness of a type A can never depend on a variable or an assumption. But that property will change when going to DHOL, which is why we allow Γ to alternate between variables and assumptions.

Types A are either user-declared types a, the built-in base type bool, or function types $A \to B$. Terms are constants c, variables x, λ-abstractions $\lambda x{:}A.\, t$, function applications $f\ t$, or obtained from the built-in bool-valued connectives $=_A$ or $\Rightarrow$. As usual [1], this suffices to define all the usual quantifiers and connectives true, false, $\neg$, $\wedge$, $\vee$, $\forall$ and $\exists$. This includes $\Rightarrow$, but we make it a primitive here because we will change it in DHOL. As usual, $E[{}^{x}\!/_{t}]$ denotes the capture-avoiding substitution of the variable x with the term t within expression E.

The type and proof system uses the judgments given below. Note that we need a meta-level judgment for the equality of types because $\equiv$ is not a bool-valued connective. On the contrary, the equality of terms $\vdash s =_A t$ is a special case of the validity judgment $\vdash F$. In HOL, $\equiv$ is trivial, and the judgment is redundant. But we include it here already because it will become non-trivial in DHOL.

Name	Judgment	Intuition
theories	$\vdash T\ \mathsf{Thy}$	T is well-formed theory
contexts	$\vdash_T \Gamma\ \mathsf{Ctx}$	Γ is well-formed context
types	$\Gamma \vdash_T A\,\mathsf{tp}$	A is well-formed type
typing	$\Gamma \vdash_T t : A$	t is a well-formed term of type well-formed type A
validity	$\Gamma \vdash_T F$	well-formed Boolean F is provable
equality of types	$\Gamma \vdash_T A \equiv B$	well-formed types A and B are equal

The rules are given in Fig. 1. We assume that all names in a theory or a context are unique without making that explicit in the rules. Following common practice, we further assume that HOL types are non-empty.

Theories and contexts:

$$\frac{}{\vdash \circ\ \mathsf{Thy}} \qquad \frac{\vdash T\ \mathsf{Thy}}{\vdash T,\ a\ \mathsf{tp}\ \mathsf{Thy}} \qquad \frac{\vdash_T A\ \mathsf{tp}}{\vdash T,\ c : A\ \mathsf{Thy}} \qquad \frac{\vdash_T F : \mathsf{bool}}{\vdash T,\ c : F\ \mathsf{Thy}}$$

$$\frac{\vdash T\ \mathsf{Thy}}{\vdash_T .\ \mathsf{Ctx}} \qquad \frac{\Gamma \vdash_T A\ \mathsf{tp}}{\vdash_T \Gamma,\ x : A\ \mathsf{Ctx}} \qquad \frac{\Gamma \vdash_T F : \mathsf{bool}}{\vdash_T \Gamma,\ x : F\ \mathsf{Ctx}}$$

Lookup in theory and context:

$$\frac{a : \mathsf{tp}\ \text{in}\ T \qquad \vdash_T \Gamma\ \mathsf{Ctx}}{\Gamma \vdash_T a\ \mathsf{tp}} \qquad \frac{c : A'\ \text{in}\ T \qquad \Gamma \vdash_T A' \equiv A}{\Gamma \vdash_T c : A} \qquad \frac{c : F\ \text{in}\ T \qquad \vdash_T \Gamma\ \mathsf{Ctx}}{\Gamma \vdash_T F}$$

$$\frac{x : A'\ \text{in}\ \Gamma \qquad \Gamma \vdash_T A' \equiv A}{\Gamma \vdash_T x : A} \qquad \frac{x : F\ \text{in}\ \Gamma \qquad \vdash_T \Gamma\ \mathsf{Ctx}}{\Gamma \vdash_T F}$$

Well-formedness and equality of types:

$$\frac{\vdash_T \Gamma\ \mathsf{Ctx}}{\Gamma \vdash_T \mathsf{bool}\ \mathsf{tp}} \qquad \frac{\Gamma \vdash_T A\ \mathsf{tp} \qquad \Gamma \vdash_T B\ \mathsf{tp}}{\Gamma \vdash_T A \to B\ \mathsf{tp}} \qquad \frac{\Gamma \vdash_T A\ \mathsf{tp}}{\Gamma \vdash_T A \equiv A} \qquad \frac{\Gamma \vdash_T A \equiv A' \qquad \Gamma \vdash_T B \equiv B'}{\Gamma \vdash_T A \to B \equiv A' \to B'}$$

Typing:

$$\frac{\Gamma,\ x : A \vdash_T t : B}{\Gamma \vdash_T (\lambda x : A.\ t) : A \to B} \qquad \frac{\Gamma \vdash_T f : A \to B \qquad \Gamma \vdash_T t : A}{\Gamma \vdash_T f\ t : B} \qquad \frac{\Gamma \vdash_T s : A \qquad \Gamma \vdash_T t : A}{\Gamma \vdash_T s =_A t : \mathsf{bool}}$$

Term equality: congruence, reflexivity, symmetry, β, η

$$\frac{\Gamma \vdash_T A \equiv A' \qquad \Gamma,\ x : A \vdash_T t =_B t'}{\Gamma \vdash_T \lambda x : A.\ t =_{A \to B} \lambda x : A'.\ t'} \qquad \frac{\Gamma \vdash_T t =_A t' \qquad \Gamma \vdash_T f =_{A \to B} f'}{\Gamma \vdash_T f\ t =_B f'\ t'}$$

$$\frac{\Gamma \vdash_T t : A}{\Gamma \vdash_T t =_A t} \qquad \frac{\Gamma \vdash_T t =_A s}{\Gamma \vdash_T s =_A t} \qquad \frac{\Gamma \vdash_T (\lambda x : A.\ s)\ t : B}{\Gamma \vdash_T (\lambda x : A.\ s)\ t =_B s[{}^{x}/_{t}]} \qquad \frac{\Gamma \vdash_T t : A \to B \qquad x\ \text{not in}\ \Gamma}{\Gamma \vdash_T t =_{A \to B} \lambda x : A.\ t\ x}$$

Rules for implication:

$$\frac{\Gamma \vdash_T F : \mathsf{bool} \qquad \Gamma \vdash_T G : \mathsf{bool}}{\Gamma \vdash_T F \Rightarrow G : \mathsf{bool}} \qquad \frac{\Gamma \vdash_T F : \mathsf{bool} \qquad \Gamma,\ x : F \vdash_T G}{\Gamma \vdash_T F \Rightarrow G} \qquad \frac{\Gamma \vdash_T F \Rightarrow G \qquad \Gamma \vdash_T F}{\Gamma \vdash_T G}$$

Congruence for validity, Boolean extensionality, and non-emptiness of types:

$$\frac{\Gamma \vdash_T F =_{\mathsf{bool}} F' \qquad \Gamma \vdash_T F'}{\Gamma \vdash_T F} \qquad \frac{\Gamma \vdash_T p\ \mathsf{true} \qquad \Gamma \vdash_T p\ \mathsf{false}}{\Gamma, x : \mathsf{bool} \vdash_T p\ x} \qquad \frac{\Gamma \vdash_T F : \mathsf{bool} \qquad \Gamma, x : A \vdash_T F}{\Gamma \vdash_T F}$$

Fig. 1. HOL Rules

3 Dependent Function Types

3.1 Language

We have carefully defined HOL in such a way that only a few surgical changes are needed to define DHOL. A consolidated summary of DHOL is given in Appendix A.2 in the extended preprint [20]. The **grammar** is as follows with unchanged parts shaded out:

$$\begin{array}{lcll}
T & ::= & \circ \mid T, a : (\Pi x{:}A.\ )^{*}\mathsf{tp} \mid T, c : A \mid T, c : F & \text{theories} \\
\Gamma & ::= & . \mid \Gamma, x : A \mid \Gamma, ass : F & \text{contexts} \\
A, B & ::= & a\ t_1 \ldots t_n \mid \Pi x{:}A.\ B \mid \mathsf{bool} & \text{types} \\
s,t,f,F,G & ::= & c \mid x \mid \lambda x{:}A.\ t \mid f\ t \mid s =_A t \mid F \Rightarrow G & \text{terms}
\end{array}$$

Concretely, base types a may now take term arguments and simple function types $A \to B$ are replaced with dependent function types $\Pi x{:}A.\ B$. As usual we will retain the notation $A \to B$ for the latter if x does not occur free in B. DHOL is a conservative extension of HOL, and we recover HOL as the fragment of DHOL in which all base types a have arity 0.

Example 1 (Category Theory). As a running example, we formalize the theory of a category in DHOL. It declares the base type *obj* for objects and the dependent base type $\mathtt{mor}\ a\ b$ for morphisms. Further it declares the constants *id* and `comp` for identity and composition, and the axioms for neutrality. We omit the associativity axiom for brevity.

$$\begin{aligned}
\mathtt{obj} :& \mathtt{tp} \\
\mathtt{mor} :& \Pi x, y{:}\mathtt{obj}.\ \mathtt{tp} \\
\mathtt{id} :& \Pi a{:}\mathtt{obj}.\ \mathtt{mor}\ a\ a \\
\mathtt{comp} :& \Pi a, b, c{:}\mathtt{obj}.\ \mathtt{mor}\ a\ b \to \mathtt{mor}\ b\ c \to \mathtt{mor}\ a\ c \\
\mathtt{neutL} :& \forall x, y : \mathtt{obj}. \forall m : \mathtt{mor}\ x\ y.\ \ m \circ \mathtt{id}_x =_{\mathtt{mor}\ x\ y} m \\
\mathtt{neutR} :& \forall x, y : \mathtt{obj}.\ \forall m : \mathtt{mor}\ x\ y.\ \ \mathtt{id}_y \circ m =_{\mathtt{mor}\ x\ y} m
\end{aligned}$$

Here we use a few intuitive notational simplifications such as writing $\Pi x, y{:}\mathtt{obj}.$ for binding two variables of the same type. We also use the notations $\mathtt{id}_x$ for $\mathtt{id}\ x$ and $h \circ g$ for $\mathtt{comp}\ \_\ \_\ \_\ g\ h$ where the _ denote inferable arguments of type `obj`.

The **judgments** stay the same and we only make minor changes to the **rules**, which we explain in the sequel. Firstly we replace all rules for → with the ones for Π:

$$\frac{\Gamma \vdash_T A\ \mathsf{tp} \quad \Gamma, x : A \vdash_T B\ \mathsf{tp}}{\Gamma \vdash_T \Pi x{:}A.\ B\ \mathsf{tp}} \qquad \frac{\Gamma \vdash_T A \equiv A' \quad \Gamma, x : A \vdash_T B \equiv B'}{\Gamma \vdash_T \Pi x{:}A.\ B \equiv \Pi x{:}A'.\ B'}$$

$$\frac{\Gamma, x : A \vdash_T t : B}{\Gamma \vdash_T (\lambda x{:}A.\ t) : \Pi x{:}A.\ B} \qquad \frac{\Gamma \vdash_T f : \Pi x{:}A.\ B \qquad \Gamma \vdash_T t : A}{\Gamma \vdash_T f\ t : B[^{x}/_{t}]}$$

$$\frac{\Gamma \vdash_T A \equiv A' \ \ \Gamma, x : A \vdash_T t =_B t'}{\Gamma \vdash_T \lambda x{:}A.\ t =_{\Pi x A.\ B} \lambda x{:}A'.\ t'} \qquad \frac{\Gamma \vdash_T t =_A t' \quad \Gamma \vdash_T f =_{\Pi x A.\ B} f'}{\Gamma \vdash_T f\ t =_B f'\ t'}$$

$$\frac{\Gamma \vdash_T t : \Pi x{:}A.\, B}{\Gamma \vdash_T t =_{\Pi x A.\, B} \lambda x{:}A.\, t\, x}$$

Then we replace the rules for declaring, using, and equating base types with the ones where base types are applied to arguments:

$$\frac{\vdash_T x_1 : A_1, \dots, x_n : A_n \ \mathsf{Ctx}}{\vdash T,\ a : \Pi x_1{:}A_1.\ \dots \Pi x_n{:}A_n.\ \mathsf{tp}\ \mathsf{Thy}}$$

$$\frac{\begin{array}{c}\vdash_T \Gamma\ \mathsf{Ctx} \quad a : \Pi x_1{:}A_1.\ \dots \Pi x_n{:}A_n.\ \mathsf{tp}\ \mathsf{in}\ T \\ \Gamma \vdash_T t_1 : A_1 \ \dots\ \Gamma \vdash_T t_n : A_n[x_1/t_1]\dots[x_{n-1}/t_{n-1}]\end{array}}{\Gamma \vdash_T a\, t_1 \ \dots\ t_n\ \mathsf{tp}}$$

$$\frac{\begin{array}{c}\vdash_T \Gamma\ \mathsf{Ctx} \quad a : \Pi x_1{:}A_1.\ \dots \Pi x_n{:}A_n.\ \mathsf{tp}\ \mathsf{in}\ T \\ \Gamma \vdash_T s_1 =_{A_1} t_1 \ \dots\ \Gamma \vdash_T s_n =_{A_n[x_1/t_1]\dots[x_{i-1}/t_{i-1}]} t_n\end{array}}{\Gamma \vdash_T a\, s_1 \ \dots\ s_n \equiv a\, t_1 \ \dots t_n}$$

The last of these is the critical rule via which term equality leaks into type equality. Thus, typing of expressions may now depend on equality assumptions and thus typing becomes undecidable.

Example 2 (Undecidability of Typing). Continuing Example 1, consider terms $\vdash f : \mathtt{mor}\ u\ v$ and $\vdash g : \mathtt{mor}\ v'\ w$ for terms $\vdash u, v, v', w : \mathtt{obj}$. Then $\vdash g \circ f : \mathtt{mor}\ u\ w$ holds iff $\vdash f : \mathtt{mor}\ u\ v'$, which holds iff $\vdash v =_{\mathtt{obj}} v'$. Depending on the axioms present, this may be arbitrarily difficult to prove.

Finally, we modify the rule for the non-emptiness of types: we allow the existence of empty dependent types and only require that for each HOL type in the image of the translation there exists one non-empty DHOL type translated to it (rather than requiring all dependent types translated to it to be non-empty). And we replace the typing rule for implication with the dependent one. The proof rules for implications are unchanged.

$$\frac{\Gamma \vdash_T F : \mathsf{bool} \qquad \Gamma, x{:}F \vdash_T G : \mathsf{bool}}{\Gamma \vdash_T F \Rightarrow G : \mathsf{bool}}$$

Example 3 (Dependent Implication). Continuing Example 1, consider the formula

$$x : \mathtt{obj},\ y : \mathtt{obj} \vdash x =_{\mathtt{obj}} y \Rightarrow \mathtt{id}_x =_{\mathtt{mor}\ x\ x} \mathtt{id}_y \ :\ \mathsf{bool}$$

which expresses that equal objects have equal identity morphisms. It is easy to prove. But it is only well-typed because the typing rule for dependent implication allows using $x =_{\mathtt{obj}} y$ while type-checking $\mathtt{id}_x =_{\mathtt{mor}\ x\ x} \mathtt{id}_y : \mathsf{bool}$, which requires deriving $\mathtt{id}_y : \mathtt{mor}\ x\ x$ and thus $\mathtt{mor}\ y\ y \equiv \mathtt{mor}\ x\ x$.

All the usual connectives and quantifiers can be defined in any of the usual ways now. However, the details matter for the dependent versions of the connectives. In particular, we choose $F \wedge G := \neg(F \Rightarrow \neg G)$ and $F \vee G := \neg F \Rightarrow G$ in order to obtain the dependent versions of conjunction and disjunction, in which the well-formedness of G may depend on the truth or falsity of F, respectively.

3.2 Translation

We define a translation function $X \mapsto \overline{X}$ that maps any DHOL-syntax X to HOL-syntax. Its intuition is to erase type dependencies by translating all types $a t_1 \ldots, t_n$ to a and replacing every Π with $\rightarrow$. To recover the information of the erased dependencies, we additionally define a partial equivalence relation (PER) A^* on $\overline{A}$ for every DHOL-type A.

In general, a PER r on type U is a symmetric and transitive relation on U. This is equivalent to r being an equivalence relation on a subtype of U. The intuitive meaning of our translation is that the DHOL-type A corresponds in HOL to the quotient of the appropriate subtype of $\overline{A}$ by the equivalence A^*. In particular, the predicate $A^*\ t\ t$ captures whether t represents a term of type A. More formally, the correspondence is:

DHOL	HOL
type A	type $\overline{A}$ and PER $A^* : \overline{A} \rightarrow \overline{A} \rightarrow \mathsf{bool}$
term $t : A$	term $\overline{t} : \overline{A}$ satisfying $A^*\ \overline{t}\ \overline{t}$

Definition 1 (Translation). *We translate DHOL-syntax by induction on the grammar. Theories and contexts are translated declaration-wise:*

$$\overline{\circ} := \circ \qquad \overline{T, D} := \overline{T}, \overline{D} \qquad \overline{.} := . \qquad \overline{T, D} := \overline{T}, \overline{D}$$

where $\overline{D}$ is a list of declarations.

The translation $\overline{a : \Pi x_1 : A_1. \ldots \Pi x_n : A_n.\ \mathsf{tp}}$ of a base type declaration is given by

$$a : \mathsf{tp}, \qquad a^* : \overline{A_1} \rightarrow \ldots \rightarrow \overline{A_n} \rightarrow a \rightarrow a \rightarrow \mathsf{bool}$$

$$a_{PER} : \forall x_1 : \overline{A_1}. \ldots \forall x_n : \overline{A_n}. \forall u, v : a.\ a^*\ x_1\ \ldots\ x_n\ u\ v \Rightarrow u =_a v$$

Thus, a is translated to a base type of the same name without arguments and a trivial PER for every argument tuple. Intuitively, $a^*\ \overline{t_1}\ \ldots\ \overline{t_n}\ u\ u$ defines the subtype of the HOL-type a corresponding to the DHOL-type $a\ t_1\ \ldots\ t_n$.

Constant and variable declarations are translated by adding the assumptions that they are in the PER of their type, and axioms and assumptions are translated straightforwardly:

$$\overline{c : A} := c : \overline{A},\ c^* : A^*\ c\ c \qquad \overline{x : A} := x : \overline{A},\ x^* : A^*\ x\ x$$

$$\overline{c\colon F} := c\colon\overline{F} \qquad \overline{x\colon F} := x\colon\overline{F}$$

The cases of $\overline{A}$ and A^* for types A are:

$$\overline{a\, t_1\, \dots\, t_n} := a \qquad (a\, t_1\, \dots\, t_n)^*\; s\, t := a^*\; \overline{t_1}\, \dots\, \overline{t_n}\; s\, t$$

$$\overline{\Pi x{:}A.\, B} := \overline{A} \to \overline{B} \qquad (\Pi x{:}A.\, B)^*\; f\, g := \forall x,y{:}\overline{A}.\; A^*\; x\, y \Rightarrow B^*\; (f\, x)\; (g\, y)$$

$$\overline{\mathsf{bool}} := \mathsf{bool} \qquad \mathsf{bool}^*\; s\, t := s =_{\mathsf{bool}} t$$

Finally, the cases for terms are straightforward except for, crucially, translating equality to the respective PER:

$$\overline{c} := c \qquad \overline{x} := x \qquad \overline{\lambda x{:}A.\, t} := \lambda x{:}\overline{A}.\, \overline{t} \qquad \overline{f\, t} := \overline{f}\, \overline{t}$$

$$\overline{F \Rightarrow G} := \overline{F} \Rightarrow \overline{G} \qquad \overline{s =_A t} := A^*\; \overline{s}\, \overline{t}$$

Example 4 (Translating Derived Connectives). If we define true, false, $\neg$ as usual in HOL and use the definition for dependent conjunction from above, it is straightforward to show that all DHOL-connectives are translated to their HOL-counterparts. For example, we have (up to logical equivalence in HOL) that $\overline{F \wedge G} = \overline{F} \wedge \overline{G}$.

We also define the quantifiers in the usual way, e.g., using $\forall x : A.F(x) := \lambda x : A.\; F(x) =_{A\to\mathsf{bool}} \lambda x{:}A.\; \mathsf{true}$. Then applying our translation yields

$$\overline{\forall x : A.F(x)} = (A \to \mathsf{bool})^*\; \overline{\lambda x : A.F(x)}\; \overline{\lambda x : A.\mathsf{true}}$$

$$= \forall x,y : \overline{A}.A^*\; x\, y \Rightarrow \mathsf{bool}^*\; F(x)\; \mathsf{true}$$

This looks clunky, but (because A^* is a PER as shown in Theorem 1) is equivalent to $\forall x : \overline{A}.A^*\; x\, x \Rightarrow F(x)$. Thus, DHOL-$\forall$ is translated to HOL-$\forall$ relativized using $A^*\; x\, x$. The corresponding rule $\overline{\exists x : A.F(x)} = \exists x : \overline{A}.A^*\; x\, x \wedge F(x)$ can be shown accordingly.

Example 5 (Categories in HOL). We give a fragment of the translation of Example 1:

$$\begin{array}{ll}
\mathtt{obj} : \mathtt{tp} & \mathtt{obj}^* : \mathtt{obj} \to \mathtt{obj} \to \mathsf{bool} \\
\mathtt{mor} : \mathtt{tp} & \mathtt{mor}^* : \mathtt{obj} \to \mathtt{obj} \to \mathtt{mor} \to \mathtt{mor} \to \mathsf{bool} \\
\mathtt{id} : \mathtt{obj} \to \mathtt{mor} & \mathtt{id}^* : \forall x,y : \mathtt{obj}.\mathtt{obj}^*\; x\, y \Rightarrow \mathtt{mor}^*\; x\, x\; (\mathtt{id}\; x)\; (\mathtt{id}\; y) \\
\multicolumn{2}{l}{\mathtt{comp} : \mathtt{obj} \to \mathtt{obj} \to \mathtt{obj} \to \mathtt{mor} \to \mathtt{mor} \to \mathtt{mor}} \\
\multicolumn{2}{l}{\mathtt{neutL} : \forall x : \mathtt{obj}.\mathtt{obj}^*\; x\, x \Rightarrow \forall y : \mathtt{obj}.\mathtt{obj}^*\; y\, y \Rightarrow} \\
\multicolumn{2}{l}{\qquad\qquad \forall m : \mathtt{mor}.\mathtt{mor}^*\; x\, y\, m\, m \Rightarrow \mathtt{mor}^*\; x\, y\; (\mathtt{comp}\; x\, x\, y\; (\mathtt{id}\; x)\; m)\; m}
\end{array}$$

Here, for brevity, we have omitted $\mathtt{obj}_{PER}$, $\mathtt{mor}_{PER}$, and $\mathtt{comp}^*$ and have already used the translation rule for $\forall$ from Example 4. The result is structurally close to what a native formalization of categories in HOL would look like, but somewhat clunkier.

Typing rules for predicate subtypes:

$$\frac{\Gamma \vdash_T p : \Pi x : A.\, \mathsf{bool}}{\Gamma \vdash_T A|_p\ \mathsf{tp}} \qquad \frac{\Gamma \vdash_T t : A \qquad \Gamma \vdash_T p\, t}{\Gamma \vdash_T t : A|_p} \qquad \frac{\Gamma \vdash_T t : A|_p}{\Gamma \vdash_T p\, t}$$

Congruence and variance rule for predicate subtypes:

$$\frac{\Gamma \vdash_T A \equiv A' \qquad \Gamma \vdash_T p =_{\Pi x:A.\ \mathsf{bool}} p'}{\Gamma \vdash_T A|_p \equiv A'|_{p'}} \qquad \frac{\Gamma \vdash_T A <: A' \qquad \Gamma,\ x : A \vdash_T p\, x \Rightarrow p'\, x}{\Gamma \vdash_T A|_p <: A'|_{p'}}$$

Rules that relate A and $A|_p$:

$$\frac{\Gamma \vdash_T A <: A'}{\Gamma \vdash_T A|_p <: A'} \qquad \frac{\Gamma \vdash_T A\ \mathsf{tp}}{\Gamma \vdash_T A \equiv A|_{\lambda x:A.\ \mathsf{true}}} \qquad \frac{\Gamma \vdash_T A\ \mathsf{tp}}{\Gamma \vdash_T A|_{\lambda x:A.\ \mathsf{true}} \equiv A}$$

Variance rules for other DHOL types:

$$\frac{\Gamma \vdash_T A \equiv A'}{\Gamma \vdash_T A <: A'} \qquad \frac{\Gamma \vdash_T A' <: A \qquad \Gamma,\ x : A' \vdash_T B <: B'}{\Gamma \vdash_T \Pi x : A.\, B <: \Pi x : A'.\, B'}$$

Rules for normalizing certain subtypes:

$$\frac{\Gamma \vdash_T A\ \mathsf{tp} \qquad \Gamma,\ x : A \vdash_T B\ \mathsf{tp} \qquad \Gamma,\ x : A \vdash_T p : \Pi y : B.\, \mathsf{bool}}{\Gamma \vdash_T \Pi x : A.\, (B|_p) \equiv (\Pi x : A.\, B)|_{\lambda f.\forall x:A.\ p\ (f\ x)}}$$

$$\frac{\Gamma \vdash_T A\ \mathsf{tp} \qquad \Gamma \vdash_T p : \Pi x : A.\, \mathsf{bool} \qquad \Gamma \vdash_T q : \Pi x : (A|_p).\, \mathsf{bool}}{\Gamma \vdash_T A|_p|_q \equiv A|_{\lambda x:A.\ p\ x \wedge q\ x}}$$

Fig. 2. Additional Rules for Predicate Subtypes

4 Predicate Subtypes

To add predicate subtypes, we extend the **grammar** with the production $A ::= A|_F$. No new productions for terms are needed because the inhabitants of $A|_F$ use the same syntax as those of A.

Example 6 (Isomorphisms). We continue Example 1 and use predicate subtypes to write the type $\mathtt{isomorphisms}\ u$ of automorphisms on u as a subtype of $\mathtt{mor}\ u\ u$. We can define $\mathtt{isomorphisms}\ u := (\mathtt{mor}\ u\ u)|_p$ where the predicate p is given by

$$\lambda m : \mathtt{mor}\ u\ u.\, \exists i : \mathtt{mor}\ u\ u.\, (i \circ m =_{\mathtt{mor}\ u\ u} \mathtt{id}_u) \wedge (m \circ i =_{\mathtt{mor}\ u\ u} \mathtt{id}_u)$$

Adding subtyping requires a few extensions to our type system. First we add a **judgment** $\Gamma \vdash_T A <: B$ and replace the lookup rules for variables and constants with their subtyping-aware variants:

$$\frac{c : A' \text{ in } T \qquad \Gamma \vdash_T A' <: A}{\Gamma \vdash_T c : A} \qquad \frac{x : A' \text{ in } \Gamma \qquad \Gamma \vdash_T A' <: A}{\Gamma \vdash_T x : A}$$

Then we add the **rules** given in Fig. 2. These induce an algorithm for deciding subtyping relative to an oracle for the undecidable validity judgment. The latter enters the algorithm when two predicate subtypes are compared. Note that the type-equality rule for $A|_p|_q$ uses a dependent conjunction.

The resulting system is a conservative extension of the variants of HOL and DHOL without subtyping: we recover these systems as the fragments that do not use $A|_p$. In particular, in that case $A <: B$ is trivial and holds iff $A \equiv B$ holds.

Finally, we extend our **translation** by adding the cases for predicate subtypes:

Definition 2 (Translation). *We extend Definition 1 with*

$$\overline{A|_p} := \overline{A} \qquad (A|_p)^*\ s\ t := A^*\ s\ t \wedge \overline{p}\ s \wedge \overline{p}\ t$$

5 Soundness and Completeness

Now we establish that our translation is faithful, i.e. sound and complete. We will use the terms *sound* and *complete* from the perspective of using a HOL-ATP for theorem proving in DHOL, e.g., *sound* means if $\overline{F}$ is a HOL-theorem, then F is a DHOL-theorem, and *complete* is the dual.[3]

The completeness theorem states that our translation preserves all DHOL-judgments. Moreover, the theorem statement clarifies the intuition behind the translations invariants:

Theorem 1 (Completeness). *We have*

if in DHOL	*then in HOL*
$\vdash T\ \mathsf{Thy}$	$\vdash \overline{T}\ \mathsf{Thy}$
$\vdash_T \Gamma\ \mathsf{Ctx}$	$\vdash_{\overline{T}} \overline{\Gamma}\ \mathsf{Ctx}$
$\Gamma \vdash_T A\ \mathsf{tp}$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{A}\ \mathsf{tp}$ *and* $\overline{\Gamma} \vdash_{\overline{T}} A^* : \overline{A} \to \overline{A} \to \mathsf{bool}$ *and* A^* *is PER*
$\Gamma \vdash_T A \equiv B$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{A} \equiv \overline{B}$ *and* $\overline{\Gamma},\ x,y : \overline{A} \vdash_{\overline{T}} A^*\ x\ y =_{\mathsf{bool}} B^*\ x\ y$
$\Gamma \vdash_T A <: B$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{A} \equiv \overline{B}$ *and* $\overline{\Gamma},\ x,y : \overline{A} \vdash_{\overline{T}} A^*\ x\ y \Rightarrow B^*\ x\ y$
$\Gamma \vdash_T t : A$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{t} : \overline{A}$ *and* $\overline{\Gamma} \vdash_{\overline{T}} A^*\ \overline{t}\ \overline{t}$
$\Gamma \vdash_T F$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{F}$

Additionally the substitution lemma holds, i.e.,

$$\Gamma,\ x : A \vdash_T t : B \ \textit{and}\ \Gamma \vdash u : A \qquad \textit{implies} \qquad \overline{\Gamma} \vdash_{\overline{T}} \overline{t[x/u]} =_{\overline{B}} \overline{t}[x/\overline{u}]$$

Proof. The proof proceeds by induction and can be found in Appendix B of the extended preprint [20].

[3] If, however, we think of our translation as an interpretation function that maps syntax to semantics, we could also justify swapping the names of the theorems.

The reverse direction is much trickier. To understand why, we look at two canaries in the coal mine that we have used to reject multiple intuitive but untrue conjectures:

Example 7 (Non-Injectivity of the Translation). Continuing Example 1, assume terms $u,v : \mathtt{obj}$ and consider the identify functions $I_u := \lambda f : \mathtt{mor}\ u\ u. f$ and $I_v := \lambda f : \mathtt{mor}\ v\ v. f$. Both are translated to the same HOL-term $\overline{I_u} = \overline{I_v} = \lambda f : \mathtt{mor}. f$ (because I_u and I_v only differ in the type indices, which are erased by our translation).

Consequently, the ill-typed DHOL-Boolean $b := I_u =_{\mathtt{mor}\ u\ u \to \mathtt{mor}\ u\ u} I_v$ is translated to the HOL-Boolean $\lambda f : \mathtt{mor}. f =_{\mathtt{mor} \to \mathtt{mor}} \lambda f : \mathtt{mor}. f$, which is not only well-typed but even a theorem.

To better understand the underlying issue we introduce the notion of *spurious* terms. The well-typed translation $\overline{t}$ of a DHOL-term t is called **spurious** if t is ill-typed (otherwise it is called *proper*). Intuitively, we should be able to use the PERs A^* to deal with spurious terms: to type-check $t : A$ in DHOL, we want to use $A^*\ \overline{t}\ \overline{t}$ in HOL. But even that is tricky:

Example 8 (Trivial PERs for Built-In Base Types). Consider the property $\mathsf{bool}^*\ x\, x$. Our translation guarantees $\mathsf{bool}^*\ \mathsf{true}\ \mathsf{true}$ and $\mathsf{bool}^*\ \mathsf{false}\ \mathsf{false}$. Thus, we can use Boolean extensionality to prove in HOL that $\forall x : \mathsf{bool}. \mathsf{bool}^*\ x\, x$, making the property trivial. In particular, we can prove $\mathsf{bool}^*\ \overline{b}\ \overline{b}$ for the spurious Boolean b from Example 7. Even worse, the property $(\Pi x{:}A.\ B)^*\ x\ x$ is trivial in this way whenever it is for B and thus for all n-ary bool-valued function types.

More generally, this degeneration effect occurs for every base type that is built into both DHOL and HOL and that is translated to itself. bool is the simplest example of that kind, and the only one in the setting described here. But reasonable language extensions like built-in base types a for numbers, strings, etc. would suffer from the same issue. This is because all of these types would come with built-in induction principles that derive a universal property from its ground instances, at which point $a^*\ x\, x$ becomes trivial.

Note, however, that the degeneration effect does *not* occur for *user-declared* base types. For example, consider a theory that declares a base type N for the natural numbers and an induction axiom for it. N would not be translated to itself but to a fresh HOL-type in whose induction axiom the quantifier $\forall$ is relativized by $N^*\ x\, x$. Consequently, $N^*\ x\, x$ is not trivial and can be used to reject spurious terms.

These examples show that we cannot expect the reverse directions of the statements in Theorem 1 to hold in general. However, we can show the following property that is sufficient to make our translation well-behaved:

Theorem 2 (Soundness). *Assume a well-formed DHOL-theory* $\vdash T\ \mathsf{Thy}$.

$$\text{If } \Gamma \vdash_T F : \mathsf{bool} \text{ and } \overline{\Gamma} \vdash_{\overline{T}} \overline{F}, \text{ then } \Gamma \vdash_T F$$

In particular, if $\Gamma \vdash_T s : A$ *and* $\Gamma \vdash_T t : A$ *and* $\overline{\Gamma} \vdash_{\overline{T}} A^*\ \overline{s}\ \overline{t}$*, then* $\Gamma \vdash\ s =_A t$.

Proof. The key idea is to transform a HOL-proof of $\overline{F}$ into one that is in the image of the translation, at which point we can read off a DHOL-proof of F. The full proof is given in Appendix B of the extended preprint [20].

Intuitively, the reverse directions of Theorem 1 holds once we establish that all involved expressions are well-typed in DHOL. Thus, we *can* use a HOL-ATP to prove DHOL-conjectures if we validate independently that the conjecture is well-typed all along. In the remainder of the section, we develop the necessary type-checking algorithm for DHOL.

Type-Checking. Inspecting the rules of DHOL, we observe that all DHOL-judgments would be decidable if we had an oracle for the validity judgment $\Gamma \vdash_T F$. Indeed, our DHOL-rules are already written in a way that essentially allows reading off a bidirectional type-checking algorithm. It only remains to split the typing judgment $\Gamma \vdash_T t : A$ into two algorithms for type-inference (which computes A from t) and type-checking (which takes t and A and returns yes or no) and to aggregate the rules for subtyping into an appropriate pattern-match.

The construction is routine, and we have implemented the resulting algorithm in our MMT/LF logical framework [12, 19].[4] The oracle for the validity judgment is provided by our translation and a theorem prover for HOL (see Sect. 6). It remains to show that whenever the algorithm calls the oracle for $\Gamma \vdash_T F$, we do in fact have that $\Gamma \vdash_T F : \mathsf{bool}$ so that Theorem 2 is applicable. Formally, we show the following:

Theorem 3. *Relative to an oracle for $\Gamma \vdash_T F$, consider a derivation of some DHOL-judgment, in which the children of each node are ordered according to the left-to-right order of the assumptions in the statement of the applied rule.*

If the oracle calls are made in depth-first order, then each such call satisfies $\Gamma \vdash_T F : \mathsf{bool}$.

Proof. We actually prove, by induction on derivations, the more general statement requires that each rule preserves the following preconditions:

Judgment	Precondition
$\vdash_T \Gamma\ \mathsf{Ctx}$	$\vdash T\ \mathsf{Thy}$
$\Gamma \vdash_T A\,\mathsf{tp}$	$\vdash_T \Gamma\ \mathsf{Ctx}$
$\Gamma \vdash_T t : A$	$\Gamma \vdash_T A\,\mathsf{tp}$ (post-condition when used as type-*inference*)
$\Gamma \vdash_T F$	$\Gamma \vdash_T F : \mathsf{bool}$
$\Gamma \vdash_T A \equiv B$ or $\Gamma \vdash_T A <: B$	$\Gamma \vdash_T A\,\mathsf{tp}$ and $\Gamma \vdash_T B\,\mathsf{tp}$

[4] The formalization of DHOL in MMT is available at https://gl.mathhub.info/MMT/LATIN2/-/blob/devel/source/logic/hol_like/dhol.mmt. The example theories given throughout this paper and a few example conjectures are available at https://gl.mathhub.info/MMT/LATIN2/-/blob/devel/source/casestudies/2023-cade.

Note that rules whose conclusion is a validity judgment can be ignored because they are replaced by the oracle anyway.

The most interesting case is the rule for $\Gamma \vdash_T a\, s_1\, \ldots\, s_n \equiv a\, t_1\, \ldots t_n$. Here, the left-to-right order of assumptions is critical because $\Gamma \vdash_T s_1 =_{A_1} t_1$ may be needed to show, e.g., $\Gamma \vdash_T s_2 =_{A_2[^{x_1}/_{t_1}]} t_2 : \mathsf{bool}$.

6 Theorem Prover Implementation

We have integrated our translation as a preprocessor to the HOL ATP LEO-III [23]. We chose this ATP because its existing preprocessor infrastructure already includes a powerful logic embedding tool [21,22].However, with a little more effort, other HOL ATPs work as well.

Furthermore, we developed a bridge between the MMT logical framework [19] and LEO-III (both of which are written in the same programming language).This allows us to use our MMT-based type-checker for DHOL with our Leo-III-based theorem prover to obtain a full-fledge implementation of DHOL. Moreover, this system can immediately use MMT's logic-independent frontend features like IDE and module system.

Alternatively, we can use LEO-III as a general purpose DHOL-ATP that accepts input in TPTP. Even though TPTP does not officially sanction DHOL as a logic, it anticipates dependent function types and already provides syntax for them (although—to our knowledge—no ATP system has made use of it so far). Concretely, TPTP represents the type $\Pi x : A.\ B$ as `!>[X:A]:B` and a base type $a\ t_1 \ldots\ t_n$ as `a @ t1 ... @ tn`. TPTP does not yet provide syntax for predicate subtypes, i.e., this approach is currently limited to the no-subtyping fragment of DHOL. But extending the TPTP syntax with predicate subtypes would be straightforward, e.g., by using `A ?| p` to represent the type $A|_p$.

The encoding of the conjecture given in Example 3 using the theory from Example 1 is given at https://gl.mathhub.info/MMT/LATIN2/-/blob/devel/source/casestudies/2023-cade/CategoryTheory/category-theory-lemmas-dhol.p (which also includes further example conjectures relative to the same theory). Running the logic embedding tool translates it into the TPTP TH0 problem given at https://gl.mathhub.info/MMT/LATIN2/-/blob/devel/source/casestudies/2023-cade/CategoryTheory/category-theory-lemmas-hol.p. Unsurprisingly, LEO-III can prove this simple theorem easily.

Practical Evaluation. In order to evaluate the practical usefulness of the translation we studied various example conjectures about function composition in set theory and category theory. We considered 5 further lemmas based on the theory in Example 1 which are written directly in TPTP and can all be proven by E, Vampire and cvc5. We also studied various harder lemmas about function composition and category theory. Those examples are written in MMT and take advantage of advanced MMT features to improve readability, such as definitions, user-defined notations, and implicit arguments that are inferred by the prover.

The examples can be found at https://gl.mathhub.info/MMT/LATIN2/-/blob/devel/source/casestudies/2023-cade. The MMT prover successfully type-checks all problems and translates them into TPTP problems to be solved by HOL ATPs.

Since LEO-III can solve none of the 6 function composition examples, we also tested other HOL ATPs on the generated TPTP problems. Running all HOL ATP provers supported at https://www.tptp.org/cgi-bin/SystemOnTPTP on the function composition problems shows that many provers can solve 3 of the problems, Vampire can solve 4 of them, and 5 out of the 6 conjectures can be solved by at least one HOL ATP.

We also studied 6 more difficult theorems about limits in category theory including the uniqueness, commutativity, and associativity of some limits. To better evaluate the usefulness of our translation, we also formalized these lemmas in native HOL (in MMT) and compared the results. Naturally, the DHOL formalization is significantly more readable and benefits from the more expressive type system that can help spot mistakes in the formalization. Running the HOL ATPs from https://www.tptp.org/cgi-bin/SystemOnTPTP on the generated TPTP problems (with 60 s timeout) yields the results in the table below (where we omit provers that proved none of the theorems in either formalization).

HOL ATP	lemma 1 proven		lemma 2 proven		lemma 3 proven	
	DHOL	native HOL	DHOL	native HOL	DHOL	native HOL
agsyHOL	yes	no	no	no	yes	no
cocATP	yes	no	no	no	no	no
cvc5	yes	yes	no	no	yes	no
cvc5-SAT	yes	no	no	no	no	no
E	yes	yes	no	no	no	yes
HOLyHammer	yes	yes	no	no	yes	yes
Lash	yes	yes	no	no	no	no
LEO-II	yes	no	no	no	no	no
Leo-III	yes	yes	no	no	no	no
Leo-III-SAT	yes	yes	no	no	no	no
Satallax	yes	yes	no	no	yes	no
Vampire	yes	yes	no	no	no	yes
Zipperpin	yes	yes	no	no	yes	yes
total	13	9	0	0	5	4

HOL ATP	lemma 4 proven		lemma 5 proven		lemma 6 proven	
	DHOL	native HOL	DHOL	native HOL	DHOL	native HOL
agsyHOL	no	no	no	no	no	no
cocATP	no	no	no	no	no	no
cvc5	no	yes	no	no	no	no
cvc5-SAT	no	no	no	no	no	no
E	no	yes	no	yes	no	yes
HOLyHammer	no	yes	no	no	no	yes
Lash	no	no	no	no	no	no
LEO-II	no	no	no	no	no	no
Leo-III	no	no	no	no	no	no
Leo-III-SAT	no	no	no	no	no	no
Satallax	no	no	yes	no	no	no
Vampire	no	yes	no	yes	no	yes
Zipperpin	no	yes	yes	yes	no	yes
total	0	5	2	3	0	4

Overall more problems generated from the native HOL formalization can be solved by some HOL ATP (5/6 compared to 3/6 for the DHOL formalization). The HOL ATPs found 25 successful proofs for the native HOL problems and 20 for the DHOL problems. This suggests that current HOL ATPs can prove native HOL problems somewhat better than their translated DHOL counterparts, but not much better. In 8 cases a prover can prove the DHOL conjecture but not the native HOL analogue, indicating that the two formalizations have different advantages.

Furthermore, our translation has so far been engineered for generality and soundness/-completeness and not for ATP efficiency. Indeed, future work has multiple options to boost the ATP performance on translated DHOL, e.g., by

- developing sufficient criteria for when simpler HOL theories can be produced
- inserting lemmas into the translated theories that guide proof search in ATPs, e.g., to speed up equality reasoning
- adding definitions to translated DHOL problems and developing better criteria when to expand them

Thus, we consider the test results to be very promising. In particular, the translation could serve as a useful basis for type-checkers and hammer tools for DHOL ITPs.

7 Conclusion and Future Work

We have combined two features of standard languages, higher-order logic HOL and dependent type theory DTT, thereby obtaining the new dependently-typed higher-order logic DHOL. Contrary to HOL, DHOL allows for *dependent* function types. Contrary to DTT, DHOL retains the simplicity of classical Booleans and standard equality.

On the downside, we have to accept that DHOL, unlike both HOL and DTT, has an undecidable type system. Further work will show how big this disadvantage weighs in practical theorem proving applications. But we anticipate that the drawback is manageable, especially if, as in our case, an implementation of DHOL is coupled tightly with a strong ATP system. We accomplish this with a sound and complete translation from DHOL into HOL that enables using existing HOL ATPs to discharge the proof obligations that come up during type-checking. We have implemented our novel translation as a TPTP-to-TPTP preprocessor for HOL ATP systems and outlined the implementation of a type-checker and hammer tool for DHOL based on the resulting prover.

Moreover, once this design is in place, it opens up the possibility to add certain type constructors to DHOL that are often requested by users but difficult to provide for system developers because they automatically make typing undecidable. We have shown an extension of DHOL with predicate subtypes as an example. Quotients, partial functions, or fixed-length lists are other examples that can be supported in future work.

We expect our translation remains sound and complete if DHOL is extended with other features underlying common HOL systems such as built-in types for numbers, the axiom of infinity, or the subtype definition principle. How to extend DHOL with a choice operator remains a question for future work — if solved, this would allow extending existing HOL ITPs to DHOL.

Acknowledgment. Chad Brown and Alexander Steen provided valuable feedback on earlier versions of this paper.

References

1. Andrews, P.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof. Academic Press, Cambridge (1986)
2. Andrews, P., Bishop, M., Issar, S., Nesmith, D., Pfenning, F., Xi, H.: TPS: a theorem-proving system for classical type theory. J. Autom. Reasoning **16**(3), 321–353 (1996)
3. Jacobs, B., Melham, T.: Translating dependent type theory into higher order logic. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 209–229. Springer, Heidelberg (1993). https://doi.org/10.1007/BFb0037108
4. Brown, C.E.: Satallax: an automatic higher-order prover. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. Lecture Notes in Computer Science, vol. 7364, pp. 111–117. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_11
5. Church, A.: A formulation of the simple theory of types. J. Symbolic Logic **5**(1), 56–68 (1940)
6. Constable, R., et al.: Implementing Mathematics with the Nuprl Development System. Prentice-Hall, Hoboken (1986)
7. Coq Development Team: The Coq Proof Assistant: Reference Manual. Technical report, INRIA (2015)
8. Coquand, T., Huet, G.: The calculus of constructions. Inf. Comput. **76**(2/3), 95–120 (1988)

9. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 378–388. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_26
10. Gordon, M.: HOL: a proof generating system for higher-order logic. In: Birtwistle, G., Subrahmanyam, P. (eds.) VLSI Specification, Verification and Synthesis, pp. 73–128. Kluwer-Academic Publishers (1988)
11. Gordon, M., Pitts, A.: The HOL logic. In: Gordon, M., Melham, T. (eds.) Introduction to HOL, Part III, pp. 191–232. Cambridge University Press (1993)
12. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. J. Assoc. Comput. Mach. **40**(1), 143–184 (1993)
13. Harrison, J.: HOL light: a tutorial introduction. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0031814
14. Martin-Löf, P.: An intuitionistic theory of types: predicative part. In: Proceedings of the 2073 Logic Colloquium, North-Holland, pp. 73–118 (1974)
15. Norell, U.: The Agda WiKi (2005). https://wiki.portal.chalmers.se/agda
16. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_217
17. Paulson, L.C.: Isabelle. LNCS, vol. 828. Springer, Heidelberg (1994). https://doi.org/10.1007/BFb0030541
18. Pfenning, F., Schürmann, C.: System description: twelf — a meta-logical framework for deductive systems. In: CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48660-7_14
19. Rabe, F.: A modular type reconstruction algorithm. ACM Trans. Comput. Logic **19**(4), 1–43 (2018)
20. Rothgang, C., Rabe, F., Benzmüller, C.: Theorem proving in dependently-typed higher-order logic - extended preprint (2023). arXiv:2305.15382
21. Steen, A.: An extensible logic embedding tool for lightweight non-classical reasoning (2022). arXiv:2203.12352
22. Steen, A.: Logic embedding tool 1.7 (2022). https://doi.org/10.5281/zenodo.6139916
23. Steen, A., Benzmüller, C.: Extensional higher-order paramodulation in Leo-III. J. Autom. Reasoning **65**(6), 775–807 (2021)
24. Sutcliffe, G.: The TPTP problem library and associated infrastructure: the FOF and CNF parts, v3.5.0. J. Autom. Reasoning **43**(4), 337–362 (2009)

Towards Fast Nominal Anti-unification of Letrec-Expressions

Manfred Schmidt-Schauß[1] and Daniele Nantes-Sobrinho[2](✉)

[1] Goethe-University Frankfurt, Frankfurt, Germany
schauss@em.uni-frankfurt.de
[2] Imperial College London, London, UK
dnantess@ic.ac.uk

Abstract. This paper describes anti-unification algorithms for computing least general generalizations of two expressions in a functional programming language with recursive let. First, by exploring a semantic approach to the problem, we argue for an improvement of the technique used in previous papers which avoids infinite chains of properly descending generalizations. Second, we present a (non-deterministic) nominal general anti-unification algorithm applicable to general expressions, which is complete, terminating and requires polynomial time. Third, we propose a specialized anti-unification algorithm applicable to two or more garbage-free ground expressions that produces a single least general generalization in polynomial time, and which can also exploit further semantically correct equivalences. Our results have potential applications in finding clones in functional programs.

Keywords: Anti-Unification · Nominal Techniques · Generalization · Functional Programming · Recursive Let

1 Introduction

Anti-unification problems (a.k.a. *generalization* problems) consist in finding a least general generalization (lgg) of two or more given expressions. This problem has interesting applications in computer science and software engineering, such as, symbolic mathematical computing [21], proof generalization [10], clone detection [8], among others; an overview is [6]. Early proposals to apply generalization for analyzing and improving programs by syntactic manipulations was given by Plotkin [12] and Reynolds [13].

We are interested in the anti-unification problem for languages with binders, such as the lambda-calculus, the pi-calculus, or the more general nominal language [11]. For instance, $\lambda x.Z$ is a generalization of the lambda-expressions $\lambda a.app(a, a)$, $\lambda a.\lambda b.a$, and $\lambda c.c$. In fact, from $\lambda x.Z$ one can retrieve any of the

Research reported in this paper is partially funded by the EPSRC Fellowship 'VeTSpec: Verified Trustworthy Software Specification' (EP/R034567/1).

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 456–473, 2023.
https://doi.org/10.1007/978-3-031-38499-8_26

three expressions in the set by considering the appropriate instance of Z (where capturing is permitted), modulo renaming of bound variables: $Z \mapsto app(x, x)$, $Z \mapsto \lambda b.x$ and $Z \mapsto x$, respectively.

In the context of languages with recursive let (letrec), techniques for solving anti-unification problems would allow, for instance, to identify the program scheme $\texttt{letr}\ b.(\lambda x.N); a.(\lambda x.M)\ \texttt{in}\ b(y)$ as a generalization of the program [1]

$$\begin{aligned}\texttt{letr}\ & even.(\lambda x.\textsf{if-else}\ (x = 0)\ (\texttt{true})\ (odd(x-1)));\\ & odd.(\lambda x.\textsf{if-else}(x = 0)(\texttt{false})(even(x-1)))\\ & \texttt{in}\ (even\ y)\end{aligned}$$

or even identify both fragments of programs as possible clones [8].

In general, and as illustrated above, reasoning and automated deduction in higher order languages often require – as a very basic operation – to identify expressions up to α-equivalence. This means expressions are identified if they are syntactically equal up to a renaming of bound variables (which represent the binding structure). In addition, one has to have in mind that the letrec construct also satisfies laws like commutativity and associativity of its environment (e.g. we could permute the environment $b.(\lambda x.N); a.(\lambda x.M)$ as $a.(\lambda x.M); b.(\lambda x.N)$ above), which will be working in combination with binding primitives (i.e., also rename the bindings within the environment obtaining, e.g., $c.(\lambda x.M'); d.(\lambda x.N')$), and they also may occur nested.

Checking expressions for α-equivalence is an operation that is often performed on large and complex expressions. Ad-hoc algorithms for checking α-equivalence of such expressions are worst-case exponential due to searching for all possible permutations and renamings. An approach to handle α-equivalence in deduction systems is to use nominal techniques [5,11], where the focus is to ease formula specification and deduction rather than speeding up α-equivalence checking. In general, checking α-equivalence with the language extended with letrec using nominal techniques is a GI-hard problem [18]. Here, we follow the nominal approach to handle binding of names and their renaming.

In [17] we have proposed a semantic approach to anti-unification based on nominal techniques which uses atom-variables, and significantly improves an existing approach [4] to anti-unification for languages with binders, since it provides a finitary set of least general generalizations. In this work we propose a simplification of this semantic approach to a nominal language extended by the letrec construct, which we call $\texttt{NLL}_X$.

Our Results. We provide a nominal anti-unification algorithm (AntiUnifLetr) for $\texttt{NLL}_X$ which preserves the good properties of our semantic approach: it is terminating, sound, computes an exponential number of generalizations (Theorem 1) and weakly complete (Theorem 2). Completeness is achieved after further specialization of the computed generalization (Theorem 3).

The observation that *garbage* might be present in letrec expressions (for example, useless bindings in environments), and that they can be avoided by a semantically correct garbage collection algorithm, allows to apply the results and methods in [18], which shows that α-equivalence and further algorithms could be considerably improved for garbage-free expressions. This leads to the design of AntiUnifNoGarbage, an anti-unification algorithm for *ground* garbage-free expressions, that is terminating, runs in polynomial time and produces one least general generalization, i.e. it is unitary (Theorem 4).

2 Preliminaries

We consider a countable infinite set of atoms $\mathbb{A}$ of (concrete) symbols a, b which we usually denote in a meta-fashion; so we can use symbols a, b also with indices (the variables in lambda-calculus). We also consider a set $\mathcal{F}$ of function symbols with arity $ar(\cdot)$, and a countably infinite set of expression-variables *Var* ranged over by X, Y. We will use mappings on atoms from $\mathbb{A}$: a *swapping* $(a\ b)$ is a bijective function that maps atom a to atom b, atom b to a, and is the identity on other atoms. We will also use finite permutations π on atoms from $\mathbb{A}$, which consists of a composition of swappings: in fact, every finite permutation π can be represented by a composition of at most $(|dom(\pi)| - 1)$ swappings, where $dom(\pi) = \{a \in \mathbb{A} \mid \pi(a) \neq a\}$. The identity permutation is denoted Id. Composition $\pi_1 \circ \pi_2$ and the inverse π^{-1} can be immediately computed, where the complexity is polynomial in the size of $dom(\pi)$.

Ground Expressions. The syntax of expressions $\bar{e}$ of the (ground) language NLL with recursive let is:

$$\bar{e} ::= a \mid \lambda a.\bar{e} \mid (f\ \bar{e}_1\ \ldots\ \bar{e}_{ar(f)}) \mid (\texttt{letr}\ a_1.\bar{e}_1; \ldots; a_n.\bar{e}_n\ \texttt{in}\ \bar{e})$$

Ground expressions are either atoms, abstractions of an atom in an expression, function application, or a letrec expression. We assume that binding atoms $a_1, \ldots, a_n$ in a letrec-expression $(\texttt{letr}\ a_1.\bar{e}_1; \ldots; a_n.\bar{e}_n\ \texttt{in}\ \bar{e})$ are pairwise distinct. Sequences of bindings $a_1.\bar{e}_1; \ldots; a_n.\bar{e}_n$ may be abbreviated as *env* (environment). The *scope* of atom a in $\lambda a.\bar{e}$ is standard: a has scope $\bar{e}$. The letr-construct has a special scoping rule: in $(\texttt{letr}\ a_1.\bar{e}_1; \ldots; a_n.\bar{e}_n\ \texttt{in}\ \bar{e})$, every atom a_i that is free in some $\bar{e}_j$ or $\bar{e}$ is bound by the environment $a_1.\bar{e}_1; \ldots; a_n.\bar{e}_n$. This defines in NLL the notion of free atoms $FA(\bar{e})$, bound atoms $BA(\bar{e})$ in expression $\bar{e}$, and all atoms $AT(\bar{e})$ that occur in $\bar{e}$. For an environment $env = \{a_1.\bar{e}_1, \ldots, a_n.\bar{e}_n\}$, we define the set of letrec-atoms as $LA(env) = \{a_1, \ldots, a_n\}$. We say *$a$ is fresh for $\bar{e}$* iff $a \notin FA(\bar{e})$, denoted as $a \# \bar{e}$.

Remark 1. The base language NLL is a lambda calculus extended with function constant and a recursive let constructor letr, and can also be interpreted as an untyped fragment of Haskell [7]. The function application operator in functional languages (implicit in some languages) can be encoded by a binary function app, and the case-construct in its plain form can be encoded as an application.

Example 1. The letrec-expression $(\texttt{letr}\ a.cons\ \bar{e}_1\ b; b.cons\ \bar{e}_2\ a\ \texttt{in}\ a)$ represents an infinite list $(cons\ \bar{e}_1\ (cons\ \bar{e}_2\ (cons\ \bar{e}_1\ (cons\ \bar{e}_2\ \ldots))))$, where $\bar{e}_1, \bar{e}_2$ are expressions and *cons* is the usual list constructor taken as a function symbol.

Syntactic α-equivalence on NLL is defined, following [16], as an extension of usual α-equivalence, where in addition the expressions $(\texttt{letr}\ a_1.\bar{e}_1;\ldots;a_n.\bar{e}_n\ \texttt{in}\ \bar{e})$ and $(\texttt{letr}\ a'_1.\bar{e}'_1;\ldots;a'_n.\bar{e}'_n\ \texttt{in}\ \bar{e}')$ are α-equivalent iff the expressions can be made equal by correctly renaming them, possibly reordering the environment.

Definition 1. *The α-equivalence $\sim_\alpha$ on $\bar{e} \in \texttt{NLL}$ is defined as follows:*

- *$a \sim_\alpha a$ for atoms a.*
- *if $\bar{e}_i \sim_\alpha \bar{e}'_i$ for all i, then $(f\ \bar{e}_1 \ldots \bar{e}_n) \sim_\alpha (f\ \bar{e}'_1 \ldots \bar{e}'_n)$ for n-ary $f \in \mathcal{F}$.*
- *If $\bar{e} \sim_\alpha \bar{e}'$, then $\lambda a.\bar{e} \sim_\alpha \lambda a.\bar{e}'$.*
- *If $a\#\bar{e}'$ and $\bar{e} \sim_\alpha (a\ b) \cdot \bar{e}'$, then $\lambda a.\bar{e} \sim_\alpha \lambda b.\bar{e}'$.*
- *$(\texttt{letr}\ a_1.\bar{e}_1;\ldots;a_n.\bar{e}_n\ \texttt{in}\ \bar{e}) \sim_\alpha (\texttt{letr}\ a_{\rho(1)}.\bar{e}_{\rho(1)};\ldots;a_{\rho(n)}.\bar{e}_{\rho(n)}\ \texttt{in}\ \bar{e})$ for any permutation ρ of $\{1,\ldots,n\}$.*
- *The following holds for a permutation π on atoms $\{a_1,\ldots,a_n\} \cup \{a'_1,\ldots,a'_n\}$:*

$$\frac{\forall i.\ \pi(a'_i) = a_i \quad \pi \cdot \bar{e}'_i \sim_\alpha e_i \quad \pi \cdot \bar{e}' \sim_\alpha \bar{e} \quad a_i\#(\texttt{letr}\ a'_1.\bar{e}'_1;\ldots;a'_n.\bar{e}'_n\ \texttt{in}\ \bar{e}')}{(\texttt{letr}\ a_1.\bar{e}_1;\ldots;a_n.\bar{e}_n\ \texttt{in}\ \bar{e}) \sim_\alpha (\texttt{letr}\ a'_1.\bar{e}'_1;\ldots;a'_n.\bar{e}'_n\ \texttt{in}\ \bar{e}')}$$

where, for $i = 1,\ldots,n$: a_i's are pairwise distinct, and a'_i's are pairwise distinct.

Permutations operate on NLL-expressions by recursing on their structure. For example, $\pi{\cdot}(\texttt{letr}\ a_1.\bar{e}_1;\ldots;a_n.\bar{e}_n\ \texttt{in}\ \bar{e}) = (\texttt{letr}\ \pi{\cdot}a_1.\pi{\cdot}\bar{e}_1;\ldots;\pi{\cdot}a_n.\pi{\cdot}\bar{e}_n\ \texttt{in}\ \pi{\cdot}\bar{e})$.

General Expressions. The syntax of the nominal higher-order language $\texttt{NLL}_X$ with letrec and variables is:

$$\begin{array}{ll} e, s, t ::= a \mid \pi{\cdot}X \mid \lambda a.e \mid (f\ e_1\ \ldots\ e_{ar(f)}) \mid (\texttt{letr}\ a_1.e_1;\ldots;a_n.e_n\ \texttt{in}\ e) \\ \pi \quad\ := \emptyset \mid (a\ b){\cdot}\pi \end{array}$$

General expressions extend NLL with *suspensions*, i.e., expressions of the form $\pi \cdot X$, which denotes a variable X (also called a generalization variable) in which a permutation is suspended: π is waiting for some instantiation of X before its action. The basic properties and functions of NLL such as $FA(e)$, $BA(e)$, scope, fresh, etc., extend to $\texttt{NLL}_X$ as expected. In particular, $AT(e)$ is extended to suspensions as $AT(\pi \cdot X) = \{a \mid a \in dom(\pi)\}$. The suspension $Id{\cdot}X$ is written simply as X. We define $Head(s)$ either as the top function symbol in $\{a, f, \lambda, \texttt{letr}\}$ or $Head(\pi \cdot X)$ as X. More generally, for a non-variable expression e, the expression $\pi{\cdot}e$ means an operation, which is performed by shifting π into the expression, using the additional simplification $\pi_1{\cdot}(\pi_2{\cdot}e) \to (\pi_1 \circ \pi_2){\cdot}e$, where after the shift, π only remains in suspensions. For instance, $(a\ c) \cdot (\texttt{letr}\ a.(\lambda b.X)\ \texttt{in}\ f(a))$ denotes a renaming of a to c and vice-versa, which is equal to $(\texttt{letr}\ c.(\lambda b.(a\ c) \cdot X)\ \texttt{in}\ f(c))$.

An $\texttt{NLL}_X$-*freshness constraint* is an expression of the form $a\#e$, expressing that a is not free in (or is fresh for) e, where e is an $\texttt{NLL}_X$-expression. A conjunction (or set) of freshness constraints is called *freshness context* which is written using the notation ∇, Δ. Every $\texttt{NLL}_X$-freshness context can be transformed into

$$\frac{\{a\#b\}\uplus\nabla}{\nabla}\ \text{if } a \neq b \qquad \frac{\{a\#(\pi\cdot X)\}\uplus\nabla}{\{\pi^{-1}(a)\#X\}\cup\nabla} \qquad \frac{\{a\#(f\ s_1\ldots s_n)\}\uplus\nabla}{\{a\#s_1,\ldots,a\#s_n\}\cup\nabla}$$

$$\frac{\{a\#(\lambda a.s)\}\uplus\nabla}{\nabla} \qquad \frac{\{a\#a\}\uplus\nabla}{\bot} \qquad \frac{\{a\#(\lambda b.s)\}\uplus\nabla}{\{a\#s\}\cup\nabla}\ \text{if } a \neq b$$

$$\frac{\{a\#(\mathtt{letr}\ a_1.s_1;\ldots,a_n.s_n\ \mathtt{in}\ r)\}\uplus\nabla}{\nabla}\ \text{if } a \in \{a_1,\ldots,a_n\}$$

$$\frac{\{a\#(\mathtt{letr}\ a_1.s_1;\ldots,a_n.s_n\ \mathtt{in}\ r)\}\uplus\nabla}{\{a\#s_1,\ldots a\#s_n, a\#r\}\cup\nabla}\ \text{if } a \notin \{a_1,\ldots,a_n\}$$

Fig. 1. Simplification of freshness constraints in $\mathtt{NLL}_X$

a simpler one (flattened form) using the rules in Fig. 1 exhaustively until consisting only of constraints of the form $a\#X$ or $\bot$ (fail), which are called *atomic*. An $\mathtt{NLL}_X$-freshness context ∇ is *consistent* if its flattened form does not contain $\bot$. The definition of α-equivalence extends to $\mathtt{NLL}_X$ as expected. In the following, $[s]_\alpha$ denotes the equivalence class of the expression s induced by the equivalence relation $\sim_\alpha$.

Lemma 1. *Simplification using rules of Fig. 1 constitutes a polynomial decision algorithm for satisfiability of ∇: If $\bot$ is in the result, then unsatisfiable; otherwise, satisfiable.*

An $\mathtt{NLL}_X$-substitution ρ is a finite mapping from generalization variables to $\mathtt{NLL}_X$-expressions. Substitutions act on expressions homomorphically and this action extends to freshness constraints and contexts as follows: $(a\#X)\rho$ iff $a\#X\rho$ and $\nabla\rho = \{a\#e\rho \mid a\#e \in \nabla\}$. We will denote the domain of substitutions by $dom(\cdot)$. A substitution is *ground* if it maps (generalization) variables to $\mathtt{NLL}$-expressions. For a ground substitution ρ: $\nabla\rho$ is called *valid* iff $\nabla\rho$ is consistent.

Permutations and Cycles. A *cycle* τ in $\mathbb{A}$ is a permutation represented by a sequence of different atoms $a_1, a_2, \ldots, a_n$, such that $\tau(a_i) = a_{i+1}$ for $i = 1,\ldots,n-1$ and $\tau(a_n) = a_1$. As standard, such cycle will be denoted as $\tau = (a_1\ a_2\ \ldots a_n)$. Every permutation π has a representation $\tau_1\tau_2\ldots\tau_n$ (which abbreviates $\tau_1 \circ \tau_2 \circ \ldots \circ \tau_n$) where τ_i are disjoint (primitive) cycles.

The disjoint cycles can be permuted. For instance, the permutation $(a\ b)(b\ d)(c\ e)$ has the cycle presentation $(a\ b\ d)(c\ e)$ which is the same as $(c\ e)(a\ b\ d)$.

2.1 Data-Structures of Anti-unification Algorithms

Anti-unification algorithms will produce as a result expressions that are restricted by a freshness context. These are called *expressions-in-context* and denoted as (∇, s), where ∇ is a freshness context and s is an $\mathtt{NLL}_X$-expression.

The semantics of expressions-in-context follow the idea that syntactically used names of atoms in expressions are fixed, and atoms occurring in ∇, but not in s are viewed as existentially quantified: these are treated as arbitrary names of atoms.

Definition 2. *An* expression-in-context *is a pair* (∇, e)*, where* e *is an expression and* ∇ *is a (consistent) freshness context. The* semantics *of* (∇, e) *is the set of ground instances of* e *that satisfy* ∇*, i.e.,*

$$[\![(\nabla, e)]\!] = \{[r]_\alpha \mid \exists \hat{\rho} : \forall a \in AT(e).\ a\hat{\rho} = a \text{ and } [r]_\alpha = [e\hat{\rho}]_\alpha \text{ and } \nabla\hat{\rho} \text{ valid}\}$$

where $\hat{\rho}$ *is a mapping from* $Var \cup \mathbb{A}$ *to ground expressions such that* $\hat{\rho}|_{\mathbb{A}}$ *is a bijection on atoms.*

The existential quantification on valid instances of expressions gives additional power to the semantics of expressions-in-context: by considering a as existentially quantified, we obtain that $[\![(\{a\#X\}, X)]\!]$ is the same as $[\![(\emptyset, X)]\!]$.

Example 2. Consider the expression-in-context $(\{a\#X\}, f(X))$. We will argue that $[\![(\{a\#X\}, f(X))]\!] = [\![(\emptyset, f(X))]\!]$. First, notice that a does not occur syntactically in $f(X)$ and therefore we can take $\hat{\rho}$ mapping a to an arbitrary atom that does not break validity of ∇. In fact:

- It is obvious that $[\![(\{a\#X\}, f(X))]\!] \subseteq [\![(\emptyset, f(X))]\!]$, since the left one has more restriction on its elements than the right one.
- $[\![(\emptyset, f(X))]\!] \subseteq [\![(\{a\#X\}, f(X))]\!]$: Let $\hat{\rho}$ be a bijection on atoms that is the identity on the atoms occurring in $f(X)$ (there is none). Then, we select $a\hat{\rho} \notin [f(X)]_\alpha$ which trivially implies that $a\hat{\rho}\#X\hat{\rho}$ holds.

Our semantics for $(\{a\#X\}, X)$ differs from the one in Baumgartner et al. [3] where $[\![(\{a\#X\}, X)]\!]_B$ is the set of all ground instances of X, where a is not permitted to occur free. This will induce the negative effect of properly infinite descending chains[1] of expressions-in-context such as $\ldots \prec_B (\{a\#X, b\#X\}, f(X)) \prec_B (\{a\#X\}, f(X)) \prec_B (\emptyset, f(X))$, which is eliminated in our approach since in all these expressions-in-context have the same semantics.

Next we define an order relation on expressions-in-context which establishes when one expression-in-context is more general or more specific than another.

Definition 3 (Ordering, Generalization).

- *An expression-in-context* (Δ, r) *is* more specific *(or less general) than an expression-in-context* (∇, s)*, denoted* $(\nabla, s) \preceq (\Delta, r)$*, if* $[\![(\Delta, r)]\!] \subseteq [\![(\nabla, s)]\!]$*. The strict part of* $\preceq$ *is denoted* $\prec$*. This defines equivalence of two expressions-in-context via their semantics:* $(\nabla, s) \approx (\nabla', t)$ *iff* $[\![(\nabla, s)]\!] = [\![(\nabla', t)]\!]$*.*
- *An expression-in-context* (Δ, r) *is a* generalization *of* (∇, s) *and* (∇', t)*, if* $(\Delta, r) \preceq (\nabla, s)$ *and* $(\Delta, r) \preceq (\nabla', t)$*.*

[1] $[\![\cdot]\!]_B$ and $\prec_B$ denote the semantics and order relation in [4], resp.

- *A generalization* (Δ', r') *of* (∇, s) *and* (∇', t) *is the* most specific *(the least general) one, if for all generalizations* (Δ, r) *of* (∇, s) *and* (∇', t)*, we have* $(\Delta, r) \preceq (\Delta', r')$*.*

For instance, the expression-in-context $(\emptyset, \lambda e.app(e, X))$ is a generalization of $(\emptyset, \lambda a.app(a, c))$ and $(\emptyset, \lambda b.app(b, Z))$, for a new atom e. It is easy to verify that $(\emptyset, \lambda e.app(e, X)) \preceq (\emptyset, \lambda a.app(a, c))$ and $(\emptyset, \lambda e.app(e, X)) \preceq (\emptyset, \lambda b.app(b, Z))$.

3 The Anti-unification Problem for $\mathtt{NLL}_X$

We are interested in *the anti-unification problem for* $\mathtt{NLL}_X$:
Given two expressions-in-context (∇, s) and (∇, t),
Find a *least general generalization*, i.e., another expression-in-context (Δ, r) that satisfies $(\Delta, r) \preceq (\nabla, s)$ and $(\Delta, r) \preceq (\nabla, t)$.

The challenge in treating letrec-expressions in anti-unification algorithms is, on the one hand, its unusual scoping and; on the other hand, the multiple possibilities to formulate the same problem in several syntactically different ways.

Remark 2 [Permutations in the generalization of suspensions]. Generalization of suspensions, say $(\emptyset, \pi_1 \cdot Z)$ and $(\emptyset, \pi_2 \cdot Z)$, need some preparations based on properties of permutations: first, we decompose π_1 and π_2 into their cycle presentation, say $\pi_1 = \mu_1 \dots \mu_n$ and $\pi_2 = \mu'_1 \dots \mu'_m$; second, we work on generalizing $(\emptyset, \mu_1 \dots \mu_n \cdot Z)$ and $(\emptyset, \mu'_1 \dots \mu'_m \cdot Z)$ as follows: let π_3 be a permutation obtained from the set of common cycles of π_1 and π_2, say $\pi_1 = \pi_3 \pi'_1$ and $\pi_2 = \pi_3 \pi'_2$. Then, $\pi_3 \cdot X$ is a generalization for $(\emptyset, \pi_1 \cdot Z)$ and $(\emptyset, \pi_2 \cdot Z)$. In the following we will denote the common cycles of permutations π_1 and π_2 as $\pi_1 \cap \pi_2$. This will be addressed in details with the specific rule for suspensions in Fig. 2.

3.1 The Algorithm AntiUnifLetr and Its Rules

We first define the nominal generalization algorithm AntiUnifLetr that (non-deterministically) computes a single generalization of the input expressions, where the generalization can also be nonlinear in the generalization variables due to merging. We will argue that the algorithm is sound and weakly complete, and one run can be performed in polynomial time.

The data structure of the algorithm AntiUnifLetr is (Γ, M, ∇, L) where:

- Γ is a set of generalization triples of the form $X : s \triangleq t$, where X is a fresh (generalization-) variable, and s, t are $\mathtt{NLL}_X$-expressions;
- M is a set of solved generalization triples;
- ∇ is a set of freshness constraints, without freshness constraints for the fresh generalization variable for the input generalization triple;
- L is a substitution represented as a set of bindings; the empty set is $[]$. The result of applying the substitution L on the generalization variable X is denoted as $X \circ L$.

We call such a tuple a *state*. The rules of the algorithm AntiUnifLetr, given in Fig. 2, operate on states and $\dot\cup$ denotes disjoint union. Given two `NLL` expressions s and t, and a freshness context Δ (possibly empty), to compute generalizations for (Δ, s) and (Δ, t), we start with $(\{X : s \triangleq t\}; \emptyset; \Delta; [])$, the *initial state* (sometimes abbreviated to $(\Delta, \{X : s \triangleq t\})$), where X is a fresh generalization variable, and we apply the rules from Fig. 2 and Fig. 4 until no more rule applications are possible and we reach the *final state* which has the form $(\emptyset, M, \nabla, L)$, where M must be completely merged. We will denote the computation from initial to a final state: $(\Gamma; \emptyset; \Delta; []) \Longrightarrow^* (\emptyset; M; \nabla, L)$.

The output is an expression-in-context obtained from the generated substitution L and the final freshness constraint ∇, i.e. the output is $(\nabla, X \circ L)$, also called the *result computed* by the AntiUnifLetr algorithm. We say it is *complete* if every least general generalization (lgg) is found and it is *weakly complete* if every lgg is found up to some set of freshness constraints.

(`Dec`): Decomposition

$$\frac{\begin{array}{c}\{X{:}f(s_1,\ldots,s_n) \triangleq f(t_1,\ldots,t_n)\} \dot\cup \Gamma, M, \nabla, L \\ X_i \text{ are fresh variables} \quad n = 0 \text{ is permitted}\end{array}}{\Gamma \cup \{X_1{:}s_1 \triangleq t_1, \ldots, X_n{:}s_n \triangleq t_n\}, M, \nabla, L \cup \{X \mapsto f(X_1,\ldots,X_n)\}}$$

(`Absaa`): Abstraction

$$\frac{\{X{:}\lambda a.s \triangleq \lambda a.t\} \dot\cup \Gamma, M, \nabla, L \qquad Y \text{is a fresh variable}}{\Gamma \cup \{Y{:}s \triangleq t\}, M, \nabla, L \cup \{X \mapsto \lambda a.Y\}}$$

(`Absab`): Abstraction

$$\frac{\{X{:}\lambda a.s \triangleq \lambda b.t\} \dot\cup \Gamma, M, \nabla, L \quad\; Y \text{ is a fresh variable} \quad c \text{ is a fresh atom}}{\Gamma \cup \{Y{:}(c\ a){\cdot}s \triangleq (c\ b){\cdot}t\}, M, \nabla \cup \{c\#\lambda a.s, c\#\lambda b.t\}\}, L \cup \{X \mapsto \lambda c.Y\}}$$

(`SusYY`): SuspensionYY

$$\frac{\begin{array}{c}\pi = \pi_1 \cap \pi_2, \pi_1 = \pi{\cdot}\pi_1', \pi_2 = \pi{\cdot}\pi_2' \\ \{X{:}\pi_1{\cdot}Y \triangleq \pi_2{\cdot}Y\} \dot\cup \Gamma, M, \nabla, L\end{array}}{\{Z{:}\pi_1'{\cdot}Y \triangleq \pi_2'{\cdot}Y\} \cup \Gamma, M, \nabla, L \cup \{X \mapsto \pi{\cdot}Z\}}$$

(`Mer`): Merging

$$\frac{\begin{array}{c}\Gamma, \{X{:}s_1 \triangleq t_1, Y{:}s_2 \triangleq t_2\} \dot\cup M, \nabla, L \\ \textsc{Eqvm}(\{(s_1,t_1) \preceq (s_2,t_2)\}) = \pi\end{array}}{\Gamma, M \cup \{X{:}s_1 \triangleq t_1\}, \nabla, L \cup \{Y \mapsto \pi{\cdot}X\}}$$

(`SolveYY`)

$$\frac{\begin{array}{c}\{X{:}\pi_1 \cdot Y \triangleq \pi_2 \cdot Y\} \dot\cup \Gamma, M, \nabla, L \\ \pi_1 \neq \pi_2, \pi_1 \cap \pi_2 = \emptyset\end{array}}{\Gamma, M \cup \{X{:}\pi_1 \cdot Y \triangleq \pi_2 \cdot Y\}, \nabla, L}$$

(`Solve`)

$$\frac{\begin{array}{c}\{X{:}s \triangleq t\} \dot\cup \Gamma, M, \nabla, L \\ \mathit{Head}(s) \neq \mathit{Head}(t) \text{ or } s, t \text{ letrec-expressions} \\ \text{with a different number of bindings}\end{array}}{\Gamma, M \cup \{X{:}s \triangleq t\}, \nabla, L}$$

Fig. 2. Rules of the algorithm AntiUnifLetr

Rules in Fig. 2 are similar to the ones in [3] without the parameter for the set of atoms occurring in the initial state and throughout the computation, and deal with abstractions, function application, and suspensions. The subalgorithm Eqvm, defined by the rules in Fig. 3, computes a matching permutation, say π, of

$$\frac{\Psi\dot{\cup}\{f(s_1,\ldots,s_n)\preceq f(s'_1,\ldots,s'_n)\}}{\Psi\cup\{s_1\preceq s'_1,\ldots,s_n\preceq s'_n\}} \qquad \frac{\Psi\dot{\cup}\{\lambda a.s\preceq\lambda a.t\}}{\Psi\cup\{s\preceq t\}}$$

$$\frac{\Psi\dot{\cup}\{\lambda a.s\preceq\lambda b.t\}\quad b\#\lambda a.s}{\Psi\cup\{(a\ b)\cdot s\preceq t\}} \qquad \frac{\Psi\dot{\cup}\{\lambda a.s\preceq\lambda b.t\}\quad a\#\lambda b.t}{\Psi\cup\{(s\preceq(a\ b)\cdot t\}}$$

1. Exhaustively apply the rules above. If after the application Ψ contains pairs not of the form $a\preceq b$, then Fail.
2. Let $\Psi=\{a_1\preceq b_1,\ldots,a_n\preceq b_n\}$. If the mapping $\{g: a_i\to b_i \mid i=1,\ldots,n,$ where $a_i\preceq b_i\in\Psi\}$ is not injective, then Fail.
3. Return the bijective mapping π generated by $a_i\to b_i, i=1,\ldots,n$.

Fig. 3. The permutation matching (sub-)algorithm Eqvm

(`Letraa`): Letrec with ordered atoms

$$\frac{\{X{:}\texttt{letr}\ a_1.s_1;\ldots;a_n.s_n\ \texttt{in}\ s\triangleq\texttt{letr}\ a_1.t_1;\ldots;a_n.t_n\ \texttt{in}\ t\}\dot{\cup}\Gamma,M,\nabla,L}{\Gamma\dot{\cup}\{X_1{:}s_1\triangleq t_1,\ldots,X_n{:}s_n\triangleq t_n,Y{:}s\triangleq t\},M,\nabla,L\dot{\cup}\{X\mapsto\texttt{letr}\ a_1.X_1,\ldots,a_n.X_n\ \texttt{in}\ Y\}}$$

(`Letperm`): Letrec with permuted bindings in environments

$$\frac{\begin{array}{c}\{X{:}\texttt{letr}\ a_1.s_1;\ldots;a_n.s_n\ \texttt{in}\ s\triangleq\texttt{letr}\ b_1.t_1;\ldots;b_n.t_n\ \texttt{in}\ t\}\dot{\cup}\Gamma,M,\nabla,L\\ \rho\text{ is a permutation on }\{1,\ldots,n\}\end{array}}{\{X:\texttt{letr}\ a_1.s_1;\ldots;a_n.s_n\ \texttt{in}\ s\triangleq\texttt{letr}\ b_{\rho(1)}.t_{\rho(1)};\ldots;b_{\rho(n)}.t_{\rho(n)}\ \texttt{in}\ t\}\cup\Gamma,M,\nabla,L}$$

(`Letrab`): Letrec with atoms in env swapped with new names

$$\frac{\begin{array}{c}\{X{:}\texttt{letr}\ a_1.s_1,\ldots,a_n.s_n\ \texttt{in}\ s\triangleq\texttt{letr}\ b_1.t_1,\ldots,b_n.t_n\ \texttt{in}\ t\}\dot{\cup}\Gamma,M,\nabla,L\\ \nabla'=\{c_i\#(\texttt{letr}\ a_1.s_1,\ldots,a_n.s_n\ \texttt{in}\ s)\}\cup\{c_i\#(\texttt{letr}\ b_1.t_1;\ldots;b_n.t_n\ \texttt{in}\ t)\}\\ \pi_1=(a_1\ c_1)\ldots(a_n\ c_n)\qquad \pi_2=(b_1\ c_1)\ldots(b_n\ c_n)\quad c_i\text{ are fresh and different atoms}\end{array}}{\{X{:}\pi_1{\cdot}(\texttt{letr}\ a_1.s_1;\ldots;a_n.s_n\ \texttt{in}\ s)\triangleq\pi_2\cdot(\texttt{letr}\ b_1.t_1;\ldots;b_n.t_n\ \texttt{in}\ t\})\dot{\cup}\Gamma,M,\nabla\dot{\cup}\nabla',L}$$

Fig. 4. Rules for `letrec` of the algorithm AntiUnifLetr

two expressions-in-context (say $s\preceq t$ in Ψ with context ∇), where EqvBiEx(Π) checks whether the set of swappings is injective and then adds a minimal set of mappings such that the result is a bijection, i.e. a permutation (on atoms). Rules in Fig. 4 are new and will be described in detail:

Rule (`Letraa`) acts as a decomposition rule with the `letr` construct and can only be applied if the bindings in the environment are the same, respecting the given order.

Rule (`Letrperm`) is branching and exhaustively tries to generalize the expressions by considering all permutations of the `letr` environment.

Rule (`Letrab`) deals with renaming of bound names; it consistently swaps the binding atoms of the `letr` environment with fresh names and propagates the obtained permutation throughout both expressions.

The latter rule exploits the following idea: if $\lambda a.s$ and $\lambda b.t$ are α-equivalent, then one can rename a and b with the same fresh name c and propagate the renaming within s and t and still obtain α-equivalent expressions.

Example 3. A generalization for the expressions-in-context $(\emptyset, \texttt{letr}\ a.a; b.c\ \texttt{in}\ f(a,b))$ and $(\emptyset, \texttt{letr}\ b.a; c.c\ \texttt{in}\ f(a,b))$ is computed as follows:

1. We cannot apply rule (`Letraa`) since the binding atoms in the environment are not corresponding to each other. We may rearrange the bindings using (`Letperm`). Then we apply rule `Letrab` for renaming: we choose d, e as fresh atoms and use the renaming $(a\ d)(b\ e)$ and $(c\ d)(b\ e)$, which leads to the check $\nabla' = \{d, e\#(\texttt{letr}\ a.a; b.c\ \texttt{in}\ f(a,b))\} \cup \{d, e\#(\texttt{letr}\ c.c; b.a\ \texttt{in}\ f(a,b))\} = \emptyset$ which holds and evaluates to $\emptyset$, since the terms are ground. After an application (`Letraa`), which decomposes the letrec environments:

$$\dfrac{\dfrac{\dfrac{(\{X : \texttt{letr}\ a.a; b.c\ \texttt{in}\ f(a,b) \triangleq \texttt{letr}\ b.a; c.c\ \texttt{in}\ f(a,b)\}, \emptyset, \emptyset, [])}{\{X{:}\texttt{letr}\ a.a; b.c\ \texttt{in}\ f(a,b) \triangleq \texttt{letr}\ c.c; b.a\ \texttt{in}\ f(a,b)\}, \emptyset, \emptyset, [])}}{(\{X{:}\texttt{letr}\ d.d; e.c\ \texttt{in}\ f(d,e) \triangleq \texttt{letr}\ d.d; e.a;\ \texttt{in}\ f(a,e)\}, \emptyset, \emptyset, [])}}{(\{X_1{:}d \triangleq d, X_2{:}c \triangleq a, Y{:}f(d,e) \triangleq f(a,e)\}, \emptyset, \emptyset, \{X \mapsto \texttt{letr}\ d.X_1; e.X_2\ \texttt{in}\ Y\})}$$

2. After three applications of (`Dec`), one (`Solve`) and one (`Mer`) we obtain $(\emptyset, \{X_2 : c \triangleq a\}, \emptyset, \{X \mapsto \texttt{letr}\ d.d; e.X_2\ \texttt{in}\ f((c\ d) \cdot X_2, e)\})$. The output generalization is $(\emptyset, \texttt{letr}\ d.d; e.X_2\ \texttt{in}\ f((c\ d) \cdot X_2, e))$.

Another Solution: from $(X : \texttt{letr}\ a.a; b.c\ \texttt{in}\ f(a,b) \triangleq \texttt{letr}\ b.a; c.c\ \texttt{in}\ f(a,b))$ we could have immediately applied the rule (`Letrab`) using $\pi_1 = (a\ d)(b\ e)$ for the left and $\pi_2 = (b\ d)(c\ e)$ for the right expression. This finally leads to a generalization of the form $\texttt{letr}\ d.X_1, e.X_2\ \texttt{in}\ f(X_3, X_4)$ which is "weaker" (too general) than the one above.

Note that the environments of one of the expressions to be generalized contains *garbage*: the binding $c.c$ is not used in $f(a,b)$.

Theorem 1. *The algorithm* AntiUnifLetr *is terminating and sound. A single run requires polynomial time. The overall computation requires exponential time and may compute an exponential number of generalizations.*

Proof. Soundness and termination can be easily checked by inspection of the rules of Figs. 2, 4 and 3. The number of nondeterministic alternatives is exponential in the worst case, and it is induced by the rule (`Letperm`). A single run (one branch) can be performed in polynomial time.

Notice that except for rule (`Letrab`), all the rules in AntiUnifLetr algorithm preserve the context ∇. This differs from the approach taken in [3] which might add new freshness constraints with a rule similar to our rule (`SolveYY`), based on a set A of all atoms appearing throughout the computation of a generalization. We show in the next example that this choice of initially preserving the freshness context leads to a weak completeness result, but completeness is regained with a specialization algorithm that will be presented next.

Example 4 (Weak Completeness). The expressions-in-context $(\emptyset, f(c_1, a))$ and $(\emptyset, f(c_2, a))$ have the generalization $(\emptyset, f(X_1, a))$ computed by the rules of Fig. 2. However, this is not the lgg since $(\{a\#X_1\}, f(X_1, a))$ is a more specific generalization. In fact, $f(a, a) \in [\![(\emptyset, f(X_1, a)]\!]$, but $f(a, a) \notin [\![(\{a\#X_1\}, f(X_1, a)]\!]$.

Theorem 2 (Weak Completeness). *Given* $\mathtt{NLL}_X$ *expressions* e *and* e'*, and a freshness context* Δ*. If* (∇', r) *is a generalization of* (Δ, e) *and* (Δ, e')*, then there exists a* ∇'' *and a derivation* $(\{X : e \triangleq e'\}, \emptyset, \Delta, []) \Longrightarrow^* (\emptyset, M, \nabla, \sigma)$ *such that* $(\nabla \cup \nabla'', X\sigma)$ *is a generalization of* (Δ, e) *and* (Δ, e') *and* $(\nabla \cup \nabla'', X\sigma) \preceq (\nabla', r)$.

Proof. The proof is by induction on the structure of r.

Example 5 (Cont. Example 4). We remark another behaviour that can be seen from the execution of AntiUnifLetr: $(\{X{:}f(c_1, a) \triangleq f(c_2, a)\}, \emptyset, \emptyset, [])$ reduces to $(\emptyset, \{X_1{:}c_1 \triangleq c_2\}, \emptyset, \{X \mapsto f(X_1, a)\})$. Notice that (i) $f(a, a)$ is clearly not an element of $[\![(\emptyset, f(c_1, a))]\!]$ nor $[\![(\emptyset, f(c_2, a))]\!]$; (ii) the information that c_1 and c_2 were free names in the input problem was "forgotten" by the generalization $f(X_1, a)$, but it can be retrieved from the solved triple in the final state. (iii) $a\#c_1$ and $a\#c_2$ hold trivially.

3.2 From Weak Completeness to Completeness

Given a result (∇, s) of a run of the algorithm AntiUnifLetr, the result is in general only weakly complete, since the expressivity of the language may permit a better generalization. The true most specific generalization may have additional freshness constraints, as it was shown in Example 4. The problem of specializing the generalizer output by AntiUnifLetr is subtle: a different but related behaviour can be seen with the next example.

Example 6. Consider the expressions-in-context $(\emptyset, f(g(c_1, a), a))$ and $(\emptyset, f(c_2, a))$ as input for AntiUnifLetr. The output generalization is $(\emptyset, f(X_1, a))$, and this is the lgg. In fact, a run of the algorithm would terminate with the final state $(\emptyset, \{X_1{:}g(c_1, a) \triangleq c_2\}, \emptyset, \{X \mapsto f(X_1, a)\})$.

We can use the information in the solved part of the final state to build the substitutions $\sigma_1 = \{X_1 \mapsto g(c_1, a)\}$ and $\sigma_2 = \{X_1 \mapsto c_2\}$ that instantiate the generalization $f(X_1, a)$ back to the input terms. Notice that $a\#X_1\sigma_1$ is equal to $a\#g(c_1, a)$ and does not hold. Thus, we cannot add $\{a\#X_1\}$ as a constraint to the generalization, since $(\{a\#X_1\}, f(X_1, a))$ cannot be instantiated to $f(g(c_1, a), a)$.

Let $\gamma = (\emptyset; M; \nabla; L)$ be a final state. We define $AT_f(\gamma)$ as the set of unbound atoms that occur in M, ∇ or $codom(L)$. We say that a generalization variable X occurs in γ when it occurs in ∇, or as a subterm in M, or in $codom(L)$.

Algorithm 1 AntiUnifLetr- Phase 2

```
Input: $(\varDelta, s)$ and $(\varDelta, t)$
$(\{X : s \triangleq t\}; \emptyset; \varDelta; []) \overset{*}{\Longrightarrow} \gamma = (\emptyset; M; \nabla; L)$
Let $(\nabla, r) = (\nabla, X \circ L)$ be the resulting generalization.
Let $X$ be a generalization variable occurring in $r$.                ▷ Repeat for each $X$
if $a \in AT(r) \backslash BA(r)$ and $a \notin RelAtoms_\gamma(X)$ then   ▷ Repeat for each $a \in AT(t)$
    $\nabla := \nabla \cup \{a\#X\}$
end if
```

Definition 4 (Relevant Atoms). *Let $\gamma = (\emptyset; M; \nabla; L)$ be a final state in a run of* AntiUnifLetr. *Let X be a generalization variable occurring in γ. The set of* relevant atoms for X, *denoted $RelAtoms_\gamma(X)$, is defined recursively:*

- *If there is no solved triple for X in M. Then, the relevant atoms are $RelAtoms_\gamma(X) = AT_f(\gamma) \backslash \{a \mid a\#X \in \nabla\}$, i.e., all atoms that are not bound and that occur syntactically in the state, but not the atoms that were excluded due to the freshness constraints in ∇.*
- *If there is a solved triple $X : s \triangleq t \in M$. Then, $RelAtoms_\gamma(X) = RelAtoms_\gamma(s) \cup RelAtoms_\gamma(t)$. The other cases are defined recursively in the structure of the expression:*
  - *$RelAtoms_\gamma(a) = a$, $RelAtoms_\gamma(f\ s_1 \dots s_n) = \bigcup_i RelAtoms_\gamma(s_i)$;*
  - *$RelAtoms_\gamma(\pi \cdot s) = \pi \cdot RelAtoms_\gamma(s)$;*
  - *$RelAtoms_\gamma(\lambda a.s) = RelAtoms_\gamma(s) \backslash \{a\}$; and*
  - *$RelAtoms_\gamma(\mathtt{letr}\ a_1.s_1; \dots; a_n.s_n\ \mathtt{in}\ r) = RelAtoms_\gamma(s_1, \dots, s_n, r) \backslash \{a_1, \dots, a_n\}$.*

For example, if we take $M = \{X{:}f(a,b) \triangleq g((a\ c) \cdot Y), Y{:}f(c,d) \triangleq g(e)\}$ and $\nabla = \{a\#Y\}$, then the set of relevant atoms for Y is $\{c,d,e\}$, and for X it is $\{a,b\} \cup (a\ c)\{c,d,e\} = \{a,b,d,e\}$, where it is noteworthy that atom c is missing.

We formulate a postprocessing algorithm (Algorithm 1) for AntiUnifLetr which is able to compute least general generalizations.

Theorem 3. *Adding (Algorithm 1) makes* AntiUnifLetr *complete.*

Note, however, that due to the non-determinism, it may be possible that one of the runs generates a generalization that is strictly less specific than the result in another run, see Example 3.

Example 7. This example shows the result of generalizing more complex expressions. Consider the generalization problem, and the sequence of generalization steps, where the last step abbreviates several steps.

$$\begin{array}{c}
(\{X_1 : \lambda a.f(a,a,c) \triangleq \lambda b.f(b,d,c)\}, \emptyset, [\ ]) \\ \hline
(\{X_1 : \lambda e.f(e,e,c) \triangleq \lambda e.f(e,d,c)\}, \emptyset, [\ ]) \\ \hline
(\{X_2 : f(e,e,c) \triangleq f(e,d,c)\}, \emptyset, \{X_1 \mapsto \lambda e.X_2\}) \\ \hline
(\emptyset, \{X_3 : e \triangleq d\}, \{X_1 \mapsto \lambda e.X_2, X_2 \mapsto f(e, X_3, c)\})
\end{array}$$

Now the resulting lgg can be computed by adding only one freshness constraint: $(\{g\#X_3\}, \lambda e.f(e, X_3, c))$. This holds, since $d \in \mathit{RelAtoms}_\gamma(X_3)$, and hence does not occur in the freshness context. Notice that $c\#X_3$ is added as a freshness constraint since c occurs in the generalization expression, but $c \notin \mathit{RelAtoms}_\gamma(X_3)$.

4 Generalization Algorithm Under Semantic Equalities

We use semantic equivalences to specialize and extend our anti-unification algorithm to *ground expressions*. In particular, we exploit the fact that removal of *garbage* is semantically correct: it does not alter the meaning of the program. First, we develop a standardization algorithm for garbage-free expressions that helps in comparing the letrec-expressions and computing generalizations in polynomial time. Second, we propose a variation of our anti-unification algorithm called AntiUnifNoGarbage.

NLL-expressions may contain irrelevant bindings in the letrec environment: for instance, in ($\texttt{letr}\ a.\texttt{Nil}; b.b\ \texttt{in}\ f(a, a)$), the binding $b.b$ is useless for the expression, and will be considered as *garbage*. The garbage bindings do not contribute to the meaning of the functional expressions. It is shown in [18], that α-equivalence of garbage-free letrec-expressions can be checked in polynomial time, and that, in general, this problem is group-isomorphism-complete [2,20].

Definition 5. *Let $\bar{e}$ be an* NLL*-expression. We say that $\bar{e}$ contains* garbage *iff there is a subexpression* ($\texttt{letr}\ a_1.\bar{e}_1, \dots, a_n.\bar{e}_n\ \texttt{in}\ \bar{e}'$) *in $\bar{e}$ such that the environment $a_1.\bar{e}_1, \dots, a_n.\bar{e}_n$ can be split into two nonempty sub-environments $a_{i_1}.\bar{e}_{i_1}, \dots, a_{i_k}.\bar{e}_{i_k}$ and $a_{j_1}.\bar{e}_{j_1}, \dots, a_{j_{k'}}.\bar{e}_{j_{k'}}$, and the binding atoms $a_{i_h}, h = i_1, \dots, i_k$ do not occur free in $\texttt{letr}\ a_{j_1}.e_{j_1}, \dots, a_{j_k}.\bar{e}_{j_k}\ \texttt{in}\ \bar{e}'$. We say that $\bar{e}$ is* garbage-free (or garbage-collected) *iff it does not contain garbage.*

Making an expression garbage-free may require an iterated removal of garbage, using the **g**arbage **r**emoval rewriting rules below:

$$\begin{aligned}
(\mathbf{gr}1)\ & \texttt{letr}\ a_1.e_1; \dots; a_n.e_n; b_1.e'_1; \dots; b_m.e'_m\ \texttt{in}\ e'_{m+1} \longrightarrow \\
& \qquad \texttt{letr}\ b_1.e'_1; \dots; b_m; e'_m\ \texttt{in}\ e'_{m+1},\ \text{ if } \bigcup \mathit{FA}(e'_i) \cap \{a_1, \dots, a_n\} = \emptyset \\
(\mathbf{gr}2)\ & \texttt{letr}\ a_1.e_1; \dots; a_n.e_n\ \texttt{in}\ e \longrightarrow e,\ \text{ if } \mathit{FA}(e) \cap \{a_1, \dots, a_n\} = \emptyset
\end{aligned}$$

We illustrate our ideas for the generalization of garbage-free expressions. Note that the used equality of expressions makes a notable difference for the results as well as for the algorithmic steps.

Example 8. Let $\bar{s} = \texttt{let}\ c.a\ \texttt{in}\ f(g(c))$ and $\bar{t} = \texttt{let}\ d.b\ \texttt{in}\ f(h(d))$ two garbage-free ground expressions. A generalization of s and t w.r.t. $\sim_\alpha$ is $\bar{s'} = \texttt{let}\ c.X_1\ \texttt{in}\ f(X_2)$, which is also an lgg. If we would allow more equalities on the expressions, like $\sim_{gc}$ as a part of the equality or even an equality $\sim_{\alpha,gc,letcp}$ that allows also copying let-bindings, then $\bar{s}$ would be equivalent to $f(g(a))$ and $\bar{t}$ equivalent to $f(h(b)$, which have $f(X)$ as a generalization. The generalisation algorithm, however, would be much more complex.

(`Letrnm`): Letrecnm   (for $m < n$)

$$\frac{\{X{:}\mathtt{letr}\ a_{n-m+1}.s_{n-m+1};\ldots;a_n.s_n\ \mathtt{in}\ s \triangleq\ \mathtt{letr}\ a_1.t_1;\ldots;a_n.t_n\ \mathtt{in}\ t\}\uplus\Gamma, M, \nabla, L}{\begin{array}{c}\{X{:}\mathtt{letr}\ a_1.a_1;\ldots;a_{n-m}.a_{n-m};a_{n-m+1}.s_{n-m+1};\ldots;a_n.s_n\ \mathtt{in}\ s\\ \triangleq \mathtt{letr}\ a_1.t_1,\ldots,a_n.t_n\ \mathtt{in}\ t\}\cup\Gamma, M, \nabla, L\end{array}}$$

Fig. 5. Different lengths of `letrec`-environments in AntiUnifLetr

The next step is to standardize the sequence of bindings in garbage-collected expressions, which greatly supports further operations.

Standardization Algorithm. Consider `let` $a_1.\bar{e}_1;\ldots;a_n.\bar{e}_n$ `in` $\bar{e}$ be a garbage-free `NLL`-expression. Then, rearrange the bindings as follows:

1. Let a_j be the atom from $\{a_1,\ldots,a_n\}$ that has the earliest occurrence as a free atom in the expression $\bar{e}$, in its printed string. Then select $a_j.\bar{e}_j$ as the leftmost binding in the fresh environment, i.e. $r_0 = \bar{e}$; $r_1 =$ `letr` $a_j.\bar{e}_j$ `in` $\bar{e}$.
2. Iterate this to compute r_k from $r_{k-1} =$ `letr` env_{k-1} `in` $\bar{e}$ by selecting among the remaining binding atoms $a_{j'} \in \{a_1,\ldots,a_n\}\backslash\{a_j\}$ again the one which first occurs free in the printed string of r_{k-1}, and then add $a_{j'}.\bar{e}_{j'}$ as the leftmost binding in the `letr`-environment obtaining $r_k =$ `letr` $a_{j'}.\bar{e}_{j'}; env_{k-1}$ `in` $\bar{e}$.

These steps are to be used iteratively: apply them to the smallest subexpression $\bar{e}'$ of $\bar{e}$, which is not yet correctly arranged. The result is a *gc-standardized* expression t_{gcst} of t.

Example 9. Consider the garbage-free expression `let` $a.app(b,\lambda c.c); b.\lambda d.d$ `in` a, where *app* is a binary function symbol for denoting the usual application of the lambda calculus. The standardization algorithm returns the gc-standardized expression `let` $b.\lambda d.d; a.app(b,\lambda c.c)$ `in` a.

Proposition 1. *For every garbage-free* `NLL`*-expression $\bar{e}$, the gc-standardized expression $\bar{e}'$ of $\bar{e}$ with $\bar{e} \sim_\alpha \bar{e}'$, has a sequence of bindings in all letrec environments that is unique and has a fixed ordering. The computation can be done in polynomial time.*

Proof. Garbage collection is polynomial: after every step the expression will be smaller, and a single step of detecting a set of redundant bindings is also polynomial. The rearrangement also can be done first for subexpressions of smaller size, and a single rearrangement of the top binding takes polynomial time.

4.1 Anti-unification of Garbage-Free Expressions

In this and the next subsection on generalization we will use a syntactically fixed ordering of bindings in a let environments, and denote this as `letf`.

AntiUnifLetr is adapted to the *ground* situation in several aspects: (i) There are no freshness constraints; (ii) expressions are first gc-standardized; (iii) we permit that $n \geq 2$ expressions are to be generalized in one step; (iv) in a set of expressions to be generalized, we make all top-level letrec environments to be of the same (minimal) length by adding bindings $a.a$ with fresh atoms a; and (v) we fix the sequence of bindings in a `let` indicated by `letf`.

We remark that an iterated generalization of pairs (i.e., to generalize s_1, s_2 and s_3 one first generalizes s_1 and s_2, and from the result, say r, one repeat the generalization process with r and s_3) has the disadvantage that from the second step, after the first application of rule, there are generalization variables, and the semantic properties get lost, which means that, e.g., the standardization is no longer usable, and so the method does no longer work properly in the next generalization steps.

Therefore, for generalizing more than 2 expressions, the data structure adopted is: the generalized state is as $(\{X{:}s_1 \triangleq \ldots \triangleq s_n\}; M; \nabla; L)$, and we use generalization tuples of the form $\{X{:}s_1 \triangleq \ldots \triangleq s_n\}$ to denote that X is a variable generalizing expressions $s_1, \ldots, s_n$. Examples for the modified rules are

$$(\mathtt{Dec_m})\ \frac{\begin{array}{c}\{X{:}f(s_{1,1},\ldots,s_{1,n}) \triangleq \ldots \triangleq f(s_{m,1},\ldots,s_{m,n})\}\dot{\cup}\Gamma, M, L\\ X_i \text{ are fresh variables} \quad n=0 \text{ is permitted}\end{array}}{\begin{array}{c}\Gamma\dot{\cup}\{X_1{:}s_{1,1}\triangleq\ldots\triangleq s_{m,1},\ldots,X_n{:}s_{1,n}\triangleq\ldots\triangleq s_{m,n}\},\\ M, L\cup\{X\mapsto f(X_1,\ldots,X_n)\}\end{array}}$$

$$(\mathtt{Absaa_m})\ \frac{\{X{:}\lambda a.s_1 \triangleq \ldots \triangleq \lambda a.s_n\}\dot{\cup}\Gamma, M, L}{\Gamma\dot{\cup}\{Y{:}s_1 \triangleq \ldots \triangleq s_n\}, M, L\cup\{X\mapsto \lambda a.Y\}}$$

$$(\mathtt{Mer_m})\ \frac{\begin{array}{c}\Gamma, \{X{:}s_1 \triangleq \ldots \triangleq s_n, Y{:}t_1 \triangleq \ldots \triangleq t_n\}\dot{\cup}M, L\\ \textsc{Eqvm}(\{(s_1,\ldots,s_n) \preceq (t_1,\ldots,t_n)\}) = \pi\end{array}}{\Gamma, M\cup\{X_1{:}s_1 \triangleq t_1\}, L\cup\{X\mapsto \pi{\cdot}Y\}}$$

Thus, we adapt the rules of AntiUnifLetr: it accepts $n \geq 2$ ground expressions; the permutation-rule (`Letrperm`) is inactive due to fixing the ordering of bindings; merging is supported, and the subalgorithms Eqvm and EqvBiEx are almost trivial and applied to larger tuples. Also the sequence of bindings in lets is fixed. All these adaptations can be done within the polynomial complexity.

These explanations suggest the algorithm AntiUnifNoGarbage, for $n \geq 2$ (ground) arguments, operating on a triple: (Γ, M, L). It is defined non-deterministically, but only one run will be done.

Example 10 (Fixed letr bindings). Generalizing the garbage-collected expressions `let` $a'.a; b'.b; c'.c$ `in` $f(g(a', b', c'))$ and `let` $a'.b; b'.c; c'.a$ `in` $f(h(a', b', c'))$ produces `let` $a'.a; b'.b; c'.c$ `in` $f(X)$ since bindings can be rearranged, which requires exponential complexity for trying rearrangements. If we fix the sequence of bindings and generalize, then the algorithm requires only polynomial time in this step, then for `letf` $a'.a; b'.b; c'.c$ `in` $f(g(a', b', c'))$ and `letf` $a'.b; b'.c; c'.a$ `in` $f(h(a', b', c'))$, we obtain `letf` $a'.X_1; b'.X_2; c'.X_3$ `in` $f(X)$.

Theorem 4. *Algorithm* AntiUnifNoGarbage *is sound, terminating and complete. It will compute a single least general generalization in polynomial time.*

Proof (Sketch). The main argument is that if no rule applies, then the result is already a generalization. Second, every applied rule keeps the semantics, i.e., does not lose information. The complexity has two components: one is the preparation of the input, which is polynomial. The second part is the test and computation of every rule, which is polynomial since there are no ∇-sets, and the execution of every rule requires polynomial time in the input size. Moreover, the size of the problem is decreased in every step.

4.2 Exploiting Semantic Equalities

Since we focus application of the algorithms in (functional) higher-order programming languages, it makes sense to take more semantic equations and properties into account to recognize semantic equality of syntactically different expressions, which improves the power of generalization algorithms.

Since there are various approaches and definitions to semantics, like variants of contextual equivalences or bisimulations [9,14,15,19] and we want to be consistent with most of them, we only investigate the equalities that are correct in a majority of the cases. By "cases" we mean different programming languages permitting `letr`, but with different operational and equational semantics.

The following semantically correct equalities, expressed as rewrite rules, in languages with letrec could also be used for further standardization of expressions, where we assume that there are no conflicts with variable names.

1. $x.f(s_1, \ldots, s_n) \to x.f(y_1, \ldots, y_n); y_1.s_1; \ldots; y_n.s_n$
2. $\texttt{let}\ (x = \texttt{letr}\ \mathit{env}\ \texttt{in}\ r); \mathit{env}'\ \texttt{in}\ s \to \texttt{let}\ x = r; \mathit{env}; \mathit{env}'\ \texttt{in}\ s$
3. $\texttt{let}\ \mathit{env}\ \texttt{in}\ (\texttt{let}\ \mathit{env}'\ \texttt{in}\ s) \to \texttt{let}\ \mathit{env}; \mathit{env}'\texttt{in}\ s.$
4. $f\ (\texttt{let}\ \mathit{env}\ \texttt{in}\ s_1)\ s_2 \to \texttt{let}\ \mathit{env}\ \texttt{in}\ (f\ s_1\ s_2).$

Note that these equalities if used to standardize expressions keep the polynomial complexity of generalizations of ground expressions.

5 Conclusion and Future Work

We formulated an anti-unification algorithm for expressions in a functional higher-order language with a let constructor that has mutually recursive bindings. We constructed a weakly complete anti-unification algorithm that in the general case is finitary, which is improved to being complete by a post-processing. In the worst case, the time for the computation as well as the number of generalizations are exponential.

In case the expressions are specialized to be ground and garbage-free, then the problem becomes unitary and the computation is polynomial. These properties make the method more friendly to applications. We also considered modifications of the generalization algorithm for functions in functional programming languages with `letr` that has a wider coverage by abstracting from the syntactical details and by observing semantic equalities.

Further work is to generalize algorithms to other patterns and to experiment with the generalization method in practice.

References

1. Ariola, Z.M., Blom, S.: Skew confluence and the lambda calculus with letrec. Ann. Pure Appl. Log. **117**(1–3), 95–168 (2002). https://doi.org/10.1016/S0168-0072(01)00104-X
2. Babai, L.: Graph isomorphism in quasipolynomial time (2016). http://arxiv.org/abs/1512.03547v2
3. Baumgartner, A., Kutsia, T., Levy, J., Villaret, M.: Nominal anti-unification. In: Fernández, M. (ed.) 26th International Conference on Rewriting Techniques and Applications, RTA 2015. LIPIcs, Warsaw, Poland, 29 June–1 July 2015, vol. 36, pp. 57–73. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPIcs.RTA.2015.57. http://www.dagstuhl.de/dagpub/978-3-939897-85-9
4. Baumgartner, A., Kutsia, T., Levy, J., Villaret, M.: Higher-order pattern anti-unification in linear time. J. Autom. Reason. **58**(2), 293–310 (2016). https://doi.org/10.1007/s10817-016-9383-3
5. Calvès, C., Fernández, M.: Matching and alpha-equivalence check for nominal terms. J. Comput. Syst. Sci. **76**(5), 283–301 (2010)
6. Cerna, D.M., Kutsia, T.: Anti-unification and generalization: a survey. https://doi.org/10.48550/arXiv.2302.00277
7. Haskell: Haskell, an advanced, purely functional programming language (2019). www.haskell.org
8. Li, H., Thompson, S.: Similar code detection and elimination for erlang programs. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 104–118. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11503-5_10
9. Moran, A., Sands, D., Carlsson, M.: Erratic fudgets: a semantic theory for an embedded coordination language. In: Ciancarini, P., Wolf, A.L. (eds.) COORDINATION 1999. LNCS, vol. 1594, pp. 85–102. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48919-3_8
10. Pfenning, F.: Unification and anti-unification in the calculus of constructions. In: Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS 1991), Amsterdam, The Netherlands, 15–18 July 1991, pp. 74–85. IEEE Computer Society (1991). https://doi.org/10.1109/LICS.1991.151632
11. Pitts, A.: Nominal techniques. ACM SIGLOG News **3**(1), 57–72 (2016). https://doi.org/10.1145/2893582.2893594
12. Plotkin, G.D.: A note on inductive generalization. Mach. Intell. **5**(1), 153–163 (1970)
13. Reynolds, J.C.: Transformational systems and algebraic structure of atomic formulas. Mach. Intell. **5**, 135–151 (1970)
14. Sabel, D., Schmidt-Schauß, M.: A contextual semantics for concurrent haskell with futures. In: Schneider-Kamp, P., Hanus, M. (eds.) Proceedings of the 13th ACM PPDP 2011, pp. 101–112. ACM (2011)
15. Sangiorgi, D., Walker, D.: On barbed equivalences in π-calculus. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 292–304. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44685-0_20

16. Schmidt-Schauß, M., Kutsia, T., Levy, J., Villaret, M., Kutz, Y.D.K.: Nominal unification and matching of higher order expressions with recursive let. Fund. Inform. **185**(3), 247–283 (2022)
17. Schmidt-Schauß, M., Nantes-Sobrinho, D.: Nominal anti-unification with atom-variables. In: Felty, A.P. (ed.) 7th FSCD 2022. LIPIcs, Haifa, Israel, vol. 228, pp. 7:1–7:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.FSCD.2022.7
18. Schmidt-Schauß, M., Rau, C., Sabel, D.: Algorithms for extended alpha-equivalence and complexity. In: van Raamsdonk, F. (ed.) 24th RTA 2013. LIPIcs, vol. 21, pp. 255–270. Schloss Dagstuhl (2013)
19. Schmidt-Schauß, M., Schütz, M., Sabel, D.: Safety of Nöcker's strictness analysis. J. Funct. Program. **18**(04), 503–551 (2008). https://doi.org/10.1017/S0956796807006624
20. Schöning, U.: Graph isomorphism is in the low hierarchy. J. Comput. Syst. Sci. **37**(3), 312–323 (1988)
21. Watt, S.M.: Algebraic generalization. SIGSAM Bull. **39**(3), 93–94 (2005). https://doi.org/10.1145/1113439.1113452

Confluence Criteria for Logically Constrained Rewrite Systems

Jonas Schöpf(✉) and Aart Middeldorp

Department of Computer Science, Universität Innsbruck, Innsbruck, Austria
{jonas.schoepf,aart.middeldorp}@uibk.ac.at

Abstract. Numerous confluence criteria for plain term rewrite systems are known. For logically constrained rewrite system, an attractive extension of term rewriting in which rules are equipped with logical constraints, much less is known. In this paper we extend the strongly-closed and (almost) parallel-closed critical pair criteria of Huet and Toyama to the logically constrained setting. We discuss the challenges for automation and present crest, a new tool for logically constrained rewriting in which the confluence criteria are implemented, together with experimental data.

Keywords: Confluence · Term Rewriting · Constraints · Automation

1 Introduction

Logically constrained rewrite systems constitute a general rewrite formalism with native support for constraints that are handled by SMT solvers. They are useful for program analysis, as illustrated in numerous papers [2,3,5,13]. Several results from term rewriting have been lifted to constrained rewriting. We mention termination analysis [6,7,12], rewriting induction [3], completion [12] as well as runtime complexity analysis [13].

In this paper we are concerned with confluence analysis of logically constrained rewrite systems (LCTRSs for short). Only two sufficient conditions for confluence of LCTRSs are known. Kop and Nishida considered (weak) orthogonality in [8]. Orthogonality is the combination of left-linearity and the absence of critical pairs, in a weakly orthogonal system trivial critical pairs are allowed. Completion of LCTRSs is the topic of [12] and the underlying confluence condition of completion is the combination of termination and joinability of critical pairs. In this paper we add two further confluence criteria. Both of these extend known conditions for standard term rewriting to the constrained setting. The first is the combination of linearity and strong closedness of critical pairs, introduced by Huet [4]. The second, also due to [4], is the combination of left-linearity and parallel closedness of critical pairs. We also consider an extension of the latter, due to Toyama [11].

This research is supported by FWF (Austrian Science Fund) Project I 5943-N.

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 474–490, 2023.
https://doi.org/10.1007/978-3-031-38499-8_27

Overview. The remainder of this paper is organized as follows. In the next section we summarize the relevant background. Section 3 recalls the existing confluence criteria for LCTRSs and some of the underlying results. The new confluence criteria for LCTRSs are reported in Sect. 4. In Sect. 5 the automation challenges we faced are described and we present our prototype implementation crest. Experimental results are reported in Sect. 6, before we conclude in Sect. 7.

2 Preliminaries

We assume familiarity with the basic notions of term rewrite systems (TRSs) [1], but shortly recapitulate terminology and notation that we use in the remainder. In particular, we recall the notion of logically constrained rewriting as defined in [3,8].

We assume a many-sorted signature $\mathcal{F}$ and a set $\mathcal{V}$ of (many-sorted) variables disjoint from $\mathcal{F}$. The signature $\mathcal{F}$ is split into term symbols from $\mathcal{F}_{\mathsf{te}}$ and theory symbols from $\mathcal{F}_{\mathsf{th}}$. The set $\mathcal{T}(\mathcal{F},\mathcal{V})$ contains the well-sorted terms over this signature and $\mathcal{T}(\mathcal{F}_{\mathsf{th}})$ denotes the set of well-sorted ground terms that consist entirely of theory symbols. We assume a mapping $\mathcal{I}$ which assigns to every sort ι occurring in $\mathcal{F}_{\mathsf{th}}$ a carrier set $\mathcal{I}(\iota)$, and an interpretation $\mathcal{J}$ that assigns to every symbol $f \in \mathcal{F}_{\mathsf{th}}$ with sort declaration $\iota_1 \times \cdots \times \iota_n \to \kappa$ a function $f_{\mathcal{J}} \colon \mathcal{I}(\iota_1) \times \cdots \times \mathcal{I}(\iota_n) \to \mathcal{I}(\kappa)$. Moreover, for every sort ι occurring in $\mathcal{F}_{\mathsf{th}}$ we assume a set $\mathcal{V}\mathsf{al}_\iota \subseteq \mathcal{F}_{\mathsf{th}}$ of value symbols, such that all $c \in \mathcal{V}\mathsf{al}_\iota$ are constants of sort ι and $\mathcal{J}$ constitutes a bijective mapping between $\mathcal{V}\mathsf{al}_\iota$ and $\mathcal{I}(\iota)$. Thus there exists a constant symbol in $\mathcal{F}_{\mathsf{th}}$ for every value in the carrier set. The interpretation $\mathcal{J}$ naturally extends to a mapping $[\![\cdot]\!]$ from ground terms in $\mathcal{T}(\mathcal{F}_{\mathsf{th}})$ to values in $\mathcal{V}\mathsf{al} = \bigcup_{\iota \in \mathcal{D}\mathsf{om}(\mathcal{I})} \mathcal{V}\mathsf{al}_\iota$: $[\![f(t_1,\ldots,t_n)]\!] = f_{\mathcal{J}}([\![t_1]\!],\ldots,[\![t_n]\!])$ for all $f(t_1,\ldots,t_n) \in \mathcal{T}(\mathcal{F}_{\mathsf{th}})$. So every ground term in $\mathcal{T}(\mathcal{F}_{\mathsf{th}})$ has a unique value. We demand that theory symbols and term symbols overlap only on values, i.e., $\mathcal{F}_{\mathsf{te}} \cap \mathcal{F}_{\mathsf{th}} \subseteq \mathcal{V}\mathsf{al}$. A term in $\mathcal{T}(\mathcal{F}_{\mathsf{th}},\mathcal{V})$ is called a *logical* term.

Positions are strings of positive natural numbers used to address subterms. The empty string is denoted by ϵ. We write $q \leqslant p$ and say that p is below q if $qq' = p$ for some position q', in which case $p \backslash q$ is defined to be q'. Furthermore, $q < p$ if $q \leqslant p$ and $q \neq p$. Finally, positions q and p are parallel, written as $q \parallel p$, if neither $q \leqslant p$ nor $p < q$. The set of positions of a term t is defined as $\mathcal{P}\mathsf{os}(t) = \{\epsilon\}$ if t is a variable or a constant, and as $\mathcal{P}\mathsf{os}(t) = \{\epsilon\} \cup \{iq \mid 1 \leqslant i \leqslant n \text{ and } q \in \mathcal{P}\mathsf{os}(t_i)\}$ if $t = f(t_1,\ldots,t_n)$ with $n \geqslant 1$. The subterm of t at position $p \in \mathcal{P}\mathsf{os}(t)$ is defined as $t|_p = t$ if $p = \epsilon$ and as $t|_p = t_i|_q$ if $p = iq$ and $t = f(t_1,\ldots,t_n)$. We write $s[t]_p$ for the result of replacing the subterm at position p of s with t. We write $\mathcal{P}\mathsf{os}_{\mathcal{V}}(t)$ for $\{p \in \mathcal{P}\mathsf{os}(t) \mid t|_p \in \mathcal{V}\}$ and $\mathcal{P}\mathsf{os}_{\mathcal{F}}(t)$ for $\mathcal{P}\mathsf{os}(t) \setminus \mathcal{P}\mathsf{os}_{\mathcal{V}}(t)$. The set of variables occurring in the term t is denoted by $\mathcal{V}\mathsf{ar}(t)$. A term t is linear if every variable occurs at most once in it. A substitution is a mapping σ from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F},\mathcal{V})$ such that its domain $\{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is finite. We write $t\sigma$ for the result of applying σ to the term t.

We assume the existence of a sort bool such that $\mathcal{I}(\mathsf{bool}) = \mathbb{B} = \{\top,\bot\}$, $\mathcal{V}\mathsf{al}_{\mathsf{bool}} = \{\mathsf{true},\mathsf{false}\}$, $[\![\mathsf{true}]\!] = \top$, and $[\![\mathsf{false}]\!] = \bot$ hold. Logical terms of sort

bool are called *constraints*. A constraint φ is *valid* if $[\![\varphi\gamma]\!] = \top$ for all substitutions γ such that $\gamma(x) \in \mathcal{V}\mathsf{al}$ for all $x \in \mathcal{V}\mathsf{ar}(\varphi)$.

A *constrained rewrite rule* is a triple $\ell \to r\ [\varphi]$ where $\ell, r \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ are terms of the same sort such that $\mathsf{root}(\ell) \in \mathcal{F}_{\mathsf{te}} \setminus \mathcal{F}_{\mathsf{th}}$ and φ is a logical term of sort bool. If $\varphi = \mathsf{true}$ then the constraint is often omitted, and the rule is denoted as $\ell \to r$. We denote the set $\mathcal{V}\mathsf{ar}(\varphi) \cup (\mathcal{V}\mathsf{ar}(r) \setminus \mathcal{V}\mathsf{ar}(\ell))$ of *logical* variables in $\rho\colon \ell \to r\ [\varphi]$ by $\mathcal{LV}\mathsf{ar}(\rho)$. We write $\mathcal{EV}\mathsf{ar}(\rho)$ for the set $\mathcal{V}\mathsf{ar}(r) \setminus (\mathcal{V}\mathsf{ar}(\ell) \cup \mathcal{V}\mathsf{ar}(\varphi))$ of variables that appear only in the right-hand side of ρ. Note that extra variables in right-hand sides are allowed, but they may only be instantiated by values. This is useful to model user input or random choice [3]. A set of constrained rewrite rules is called a *logically constrained rewrite system* (LCTRS for short).

The LCTRS $\mathcal{R}$ introduced in the example below computes the maximum of two integers.

Example 1. Before giving the rules, we need to define the term and theory symbols, the carrier sets and interpretation functions:

$$\begin{aligned}
\mathcal{F}_{\mathsf{te}} &= \{\mathsf{max}\colon \mathsf{int} \times \mathsf{int} \Rightarrow \mathsf{int}\} \cup \{\mathsf{0}, \mathsf{1}, \ldots : \mathsf{int}\} \qquad \mathcal{I}_{\mathsf{bool}} = \mathbb{B} \qquad \mathcal{I}_{\mathsf{int}} = \mathbb{Z} \\
\mathcal{F}_{\mathsf{th}} &= \{\mathsf{0}, \mathsf{1}, \ldots : \mathsf{int}\} \cup \{\mathsf{true}, \mathsf{false}\colon \mathsf{bool}\} \cup \{\neg\colon \mathsf{bool} \Rightarrow \mathsf{bool}\} \\
&\cup \{-\colon \mathsf{int} \Rightarrow \mathsf{int}\} \cup \{\wedge\colon \mathsf{bool} \times \mathsf{bool} \Rightarrow \mathsf{bool}\} \\
&\cup \{+, -\colon \mathsf{int} \times \mathsf{int} \Rightarrow \mathsf{int}\} \cup \{\leq, \geq, <, >, = : \mathsf{int} \times \mathsf{int} \Rightarrow \mathsf{bool}\}
\end{aligned}$$

The interpretations for theory symbols follow the usual semantics given in the SMT-LIB theory Ints[1] used by the SMT-LIB logic $\mathsf{QF\_LIA}$. The LCTRS $\mathcal{R}$ consists of the following constrained rewrite rules

$$\mathsf{max}(x, y) \to x\ [x \geq y] \qquad \mathsf{max}(x, y) \to y\ [y \geq x] \qquad \mathsf{max}(x, y) \to \mathsf{max}(y, x)$$

In later examples we refrain from spelling out the signature and interpretations of the theory Ints. We now define rewriting using constrained rewrite rules. LCTRSs admit two kinds of rewrite steps. Rewrite rules give rise to *rule* steps, provided the constraint of the rule is satisfied. In addition, theory calls of the form $f(v_1, \ldots, v_n)$ with $f \in \mathcal{F}_{\mathsf{th}} \setminus \mathcal{V}\mathsf{al}$ and values $v_1, \ldots, v_n$ can be evaluated in a *calculation* step. In the definition below, a substitution σ is said to *respect* a rule $\rho\colon \ell \to r\ [\varphi]$, denoted by $\sigma \vDash \rho$, if $\mathcal{D}\mathsf{om}(\sigma) = \mathcal{V}\mathsf{ar}(\ell) \cup \mathcal{V}\mathsf{ar}(r) \cup \mathcal{V}\mathsf{ar}(\varphi)$, $\sigma(x) \in \mathcal{V}\mathsf{al}$ for all $x \in \mathcal{LV}\mathsf{ar}(\rho)$, and $\varphi\sigma$ is valid. Moreover, a constraint φ is respected by σ, denoted by $\sigma \vDash \varphi$, if $\sigma(x) \in \mathcal{V}\mathsf{al}$ for all $x \in \mathcal{V}\mathsf{ar}(\varphi)$ and $\varphi\sigma$ is valid.

Definition 1. *Let $\mathcal{R}$ be an LCTRS. A* rule step $s \to_{\mathsf{ru}} t$ *satisfies $s|_p = \ell\sigma$ and $t = s[r\sigma]_p$ for some position p and constrained rewrite rule $\ell \to r\ [\varphi]$ that is respected by the substitution σ. A* calculation step $s \to_{\mathsf{ca}} t$ *satisfies $s|_p = f(v_1, \ldots, v_n)$ and $t = s[v]_p$ for some $f \in \mathcal{F}_{\mathsf{th}} \setminus \mathcal{V}\mathsf{al}$, $v_1, \ldots, v_n \in \mathcal{V}\mathsf{al}$ with $v = [\![f(v_1, \ldots, v_n)]\!]$. In this case $f(x_1, \ldots, x_n) \to y\ [y = f(x_1, \ldots, x_n)]$ with a fresh variable y is a* calculation rule. *The set of all calculation rules is denoted by $\mathcal{R}_{\mathsf{ca}}$. The relation $\to_{\mathcal{R}}$ associated with $\mathcal{R}$ is the union of $\to_{\mathsf{ru}} \cup \to_{\mathsf{ca}}$.*

[1] http://smtlib.cs.uiowa.edu/Theories/Ints.smt2.

We sometimes write $\to_{p\,|\,\rho\,|\,\sigma}$ to indicate that the rewrite step takes place at position p, using the constrained rewrite rule ρ with substitution σ.

Example 2. We have $\mathsf{max}(1+2,4) \to_{\mathcal{R}} \mathsf{max}(3,4) \to_{\mathcal{R}} \mathsf{max}(4,3) \to_{\mathcal{R}} 4$ in the LCTRS of Example 1. The first step is a calculation step. In the third step we apply the rule $\mathsf{max}(x,y) \to x\ [x \geq y]$ with substitution $\sigma = \{x \mapsto 4, y \mapsto 3\}$.

3 Confluence

In this paper we are concerned with the confluence of LCTRSs. An LCTRS $\mathcal{R}$ is *confluent* if $t \to_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^*\!\!\leftarrow u$ for all terms s, t and u such that $t\ {}_{\mathcal{R}}^*\!\!\leftarrow s \to_{\mathcal{R}}^* u$. Confluence criteria for TRSs are based on critical pairs. Critical pairs for LCTRS were introduced in [8]. The difference with the definition below is that we add dummy constraints for *extra* variables in right-hand sides of rewrite rules.

Definition 2. *An* overlap *of an LCTRS $\mathcal{R}$ is a triple $\langle \rho_1, p, \rho_2 \rangle$ with rules $\rho_1 \colon \ell_1 \to r_1\ [\varphi_1]$ and $\rho_2 \colon \ell_2 \to r_2\ [\varphi_2]$, satisfying the following conditions:*

1. *ρ_1 and ρ_2 are variable-disjoint variants of rewrite rules in $\mathcal{R} \cup \mathcal{R}_{\mathsf{ca}}$,*
2. *$p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell_2)$,*
3. *ℓ_1 and $\ell_2|_p$ are unifiable with a mgu σ such that $\sigma(x) \in \mathcal{V}\mathsf{al} \cup \mathcal{V}$ for all $x \in \mathcal{LV}\mathsf{ar}(\rho_1) \cup \mathcal{LV}\mathsf{ar}(\rho_2)$,*
4. *$\varphi_1\sigma \wedge \varphi_2\sigma$ is satisfiable, and*
5. *if $p = \epsilon$ then ρ_1 and ρ_2 are not variants, or $\mathcal{V}\mathsf{ar}(r_1) \nsubseteq \mathcal{V}\mathsf{ar}(\ell_1)$.*

In this case we call $\ell_2\sigma[r_1\sigma]_p \approx r_2\sigma\ [\varphi_1\sigma \wedge \varphi_2\sigma \wedge \psi\sigma]$ a constrained critical pair *obtained from the overlap $\langle \rho_1, p, \rho_2 \rangle$. Here*

$$\psi = \bigwedge \{x = x \mid x \in \mathcal{EV}\mathsf{ar}(\rho_1) \cup \mathcal{EV}\mathsf{ar}(\rho_2)\}$$

The set of all constrained critical pairs of $\mathcal{R}$ is denoted by $\mathsf{CCP}(\mathcal{R})$.

In the following we drop "constrained" and speak of critical pairs. The condition $\mathcal{V}\mathsf{ar}(r_1) \nsubseteq \mathcal{V}\mathsf{ar}(\ell_1)$ in the fifth condition is essential to correctly deal with extra variables in rewrite rules. The equations (ψ) added to the constraint of a critical pair save the information which variables in a critical pair were introduced by variables only occurring in the right-hand side of a rewrite rule and therefore should *only* be instantiated by values. Critical pairs as defined in [8,12] lack this information. The proof of Theorem 2 in the next section makes clear why those trivial equations are essential for our confluence criteria, see also Example 9.

Example 3. Consider the LCTRS consisting of the rule

$$\rho\colon\ \mathsf{f}(x) \to z\ [x = z\,\hat{}\,2]$$

The variable z does not occur in the left-hand side and the condition $\mathcal{V}\mathsf{ar}(r_1) \nsubseteq \mathcal{V}\mathsf{ar}(\ell_1)$ ensures that ρ overlaps with (a variant of) itself at the root position. Note that $\mathcal{R}$ is not confluent due to the non-joinable local peak $-4 \leftarrow \mathsf{f}(16) \to 4$.

Example 4. The LCTRS $\mathcal{R}$ of Example 1 admits the following critical pairs:

$$\begin{array}{ll} x \approx y \ [x \geq y \wedge y \geq x] & \langle 1, \epsilon, 2\rangle \\ x \approx \mathsf{max}(y, x) \ [x \geq y] & \langle 1, \epsilon, 3\rangle \\ y \approx \mathsf{max}(y, x) \ [y \geq x] & \langle 2, \epsilon, 3\rangle \end{array}$$

The originating overlap is given on the right, where we number the rewrite rules from left to right in Example 1.

Actually, there are three more overlaps since the position of overlap (ϵ) is the root position. Such overlaps are called *overlays* and always come in pairs. For instance, $\mathsf{max}(y, x) \approx x \ [x \geq y]$ is the critial pair originating from $\langle 3, \epsilon, 1\rangle$. For confluence criteria based on symmetric joinability conditions of critical pairs (like weak orthogonality and joinability of critical pairs for terminating systems) we need to consider just one critical pair, but this is not true for the criteria presented in the next section.

Logically constrained rewriting aims to rewrite (unconstrained) terms with constrained rules. However, for the sake of analysis, rewriting *constrained terms* is useful. In particular, since critical pairs in LCTRSs come with a constraint, confluence criteria need to consider constrained terms. The relevant notions defined below originate from [3,8].

Definition 3. *A* constrained term *is a pair $s \ [\varphi]$ of a term s and a constraint φ. Two constrained terms $s \ [\varphi]$ and $t \ [\psi]$ are* equivalent*, denoted by $s \ [\varphi] \sim t \ [\psi]$, if for every substitution γ respecting φ there is some substitution δ that respects ψ such that $s\gamma = t\delta$, and vice versa. Let $\mathcal{R}$ be an LCTRS and $s \ [\varphi]$ a constrained term. If $s|_p = \ell\sigma$ for some constrained rewrite rule $\rho\colon \ell \to r \ [\psi]$, position p, and substitution σ such that $\sigma(x) \in \mathcal{V}\mathsf{al} \cup \mathcal{V}\mathsf{ar}(\varphi)$ for all $x \in \mathcal{L}\mathcal{V}\mathsf{ar}(\rho)$, φ is satisfiable and $\varphi \Rightarrow \psi\sigma$ is valid then*

$$s \ [\varphi] \to_{\mathsf{ru}} s[r\sigma]_p \ [\varphi]$$

is a rule step. *If $s|_p = f(s_1, \ldots, s_n)$ with $f \in \mathcal{F}_{\mathsf{th}} \setminus \mathcal{F}_{\mathsf{te}}$ and $s_1, \ldots, s_n \in \mathcal{V}\mathsf{al} \cup \mathcal{V}\mathsf{ar}(\varphi)$ then*

$$s \ [\varphi] \to_{\mathsf{ca}} s[x]_p \ [\varphi \wedge x = f(s_1, \ldots, s_n)]$$

is a calculation step. *Here x is a fresh variable. We write $\to_{\mathcal{R}}$ for $\to_{\mathsf{ru}} \cup \to_{\mathsf{ca}}$ and the rewrite relation $\overset{\sim}{\to}_{\mathcal{R}}$ on constrained terms is defined as $\sim \cdot \to_{\mathcal{R}} \cdot \sim$.*

Positions in connection with $\overset{\sim}{\to}_{\mathcal{R}}$ steps always refer to the underlying steps in $\to_{\mathcal{R}}$. We give an example of constrained rewriting.

Example 5. Consider again the LCTRS $\mathcal{R}$ of Example 1. We have

$$\begin{aligned} \mathsf{max}(x + y, \mathsf{6}) \ [x \geq \mathsf{2} \wedge y \geq \mathsf{4}] &\to_{\mathcal{R}} \mathsf{max}(z, \mathsf{6}) \ [x \geq \mathsf{2} \wedge y \geq \mathsf{4} \wedge z = x + y] \\ &\to_{\mathcal{R}} z \ [x \geq \mathsf{2} \wedge y \geq \mathsf{4} \wedge z = x + y] \end{aligned}$$

The first step is a calculation step. The second step is a rule step using the rule $\mathsf{max}(x, y) \to x \ [x \geq y]$ with the substitution $\sigma = \{x \mapsto z, y \mapsto \mathsf{6}\}$. Note that the constraint $(x \geq \mathsf{2} \wedge y \geq \mathsf{4} \wedge z = x + y) \Rightarrow z \geq \mathsf{6}$ is valid.

Definition 4. *A critical pair $s \approx t\ [\varphi]$ is* trivial *if $s\sigma = t\sigma$ for every substitution σ with $\sigma \vDash \varphi$.*[2] *A left-linear LCTRS having only trivial critical pairs is called* weakly orthogonal*. A left-linear TRS without critical pairs is called* orthogonal*.*

The following result is from [8].

Theorem 1. *Weakly orthogonal LCTRS are confluent.* □

Example 6. The following left-linear LCTRS computes the Ackermann function using term symbols from $\mathcal{F}_{\mathsf{te}} = \{\mathsf{ack} : \mathsf{int} \times \mathsf{int} \Rightarrow \mathsf{int}\} \cup \{\mathsf{0}, \mathsf{1}, \cdots : \mathsf{int}\}$ and the same theory symbols, carrier sets and interpretations as in Example 1:

$$\begin{aligned}
\mathsf{ack}(\mathsf{0}, n) &\to n + \mathsf{1}\ [n \geq \mathsf{0}]\\
\mathsf{ack}(m, \mathsf{0}) &\to \mathsf{ack}(m - \mathsf{1}, \mathsf{1})\ [m > \mathsf{0}]\\
\mathsf{ack}(m, n) &\to \mathsf{ack}(m - \mathsf{1}, \mathsf{ack}(m, n - \mathsf{1}))\ [m > \mathsf{0} \wedge n > \mathsf{0}]\\
\mathsf{ack}(m, n) &\to \mathsf{0}\ [m < \mathsf{0} \vee n < \mathsf{0}]
\end{aligned}$$

Since the conjunction of any two constraints is unsatisfiable, $\mathcal{R}$ lacks critical pairs. Hence $\mathcal{R}$ is confluent by Theorem 1.

The following result is proved in [12] and forms the basis of completion of LCTRSs.

Lemma 1. *Let $\mathcal{R}$ be an LCTRS. If $t\ {}_{\mathcal{R}}\!\leftarrow s \to_{\mathcal{R}} u$ then $t \downarrow_{\mathcal{R}} u$ or $t \xleftrightarrow[\mathsf{CCP}(\mathcal{R})]{} u$.* □

In combination with Newman's Lemma, the following confluence criterion is obtained.

Corollary 1. *A terminating LCTRS is confluent if all critical pairs are joinable.*

This is less obvious than it seems. Joinability of a critical pair $s \approx t\ [\varphi]$ cannot simply be defined as $s\ [\varphi] \mathrel{\tilde{\to}}^{*}_{\mathcal{R}} \cdot {}^{*}_{\mathcal{R}}\!\mathrel{\tilde{\leftarrow}} t\ [\varphi]$, as the following example shows.

Example 7. Consider the terminating LCTRS $\mathcal{R}$ consisting of the rewrite rules

$$\mathsf{f}(x, y) \to \mathsf{g}(x, \mathsf{1} + \mathsf{1}) \qquad\qquad \mathsf{h}(\mathsf{f}(x, y)) \to \mathsf{h}(\mathsf{g}(y, \mathsf{1} + \mathsf{1}))$$

The single critical pair $\mathsf{h}(\mathsf{g}(x, \mathsf{1} + \mathsf{1})) \approx \mathsf{h}(\mathsf{g}(y, \mathsf{1} + \mathsf{1}))$ should not be joinable because $\mathcal{R}$ is not confluent, but we do have

$$\begin{aligned}
\mathsf{h}(\mathsf{g}(x, \mathsf{1} + \mathsf{1})) &\to_{\mathsf{ca}} \mathsf{h}(\mathsf{g}(x, z))\ [z = \mathsf{1} + \mathsf{1}] \sim \mathsf{h}(\mathsf{g}(y, v))\ [v = \mathsf{1} + \mathsf{1}]\\
\mathsf{h}(\mathsf{g}(y, \mathsf{1} + \mathsf{1})) &\to_{\mathsf{ca}} \mathsf{h}(\mathsf{g}(y, v))\ [v = \mathsf{1} + \mathsf{1}]
\end{aligned}$$

due to the equivalence relation $\sim$ on constrained terms; since x and y do not appear in the constraints, there is no demand that they must be instantiated with values.

[2] The triviality condition in [8] is wrong. Here we use the corrected version in an update of [8] announced on Cynthia Kop's website (accessible at https://www.cs.ru.nl/~cynthiakop/frocos13.pdf).

The solution is not to treat the two sides of a critical pair in isolation but define joinability based on rewriting constrained term pairs. So we view the symbol $\approx$ in a constrained equation $s \approx t\ [\varphi]$ as a binary constructor symbol such that the constrained equation can be viewed as a constrained term. Steps in s take place at positions $\geqslant 1$ whereas steps in t use positions $\geqslant 2$. The same is done in completion of LCTRSs [12].

Definition 5. *We call a constrained equation* $s \approx t\ [\varphi]$ trivial *if* $s\sigma = t\sigma$ *for any substitution* σ *with* $\sigma \vDash \varphi$*. A critical pair* $s \approx t\ [\varphi]$ *is* joinable *if* $s \approx t\ [\varphi] \stackrel{\sim}{\to}^{*}_{\mathcal{R}} u \approx v\ [\psi]$ *and* $u \approx v\ [\psi]$ *is trivial.*

We revisit Example 7.

Example 8. For the critical pair in Example 7 we obtain

$$\begin{aligned} &\mathsf{h}(\mathsf{g}(x,\mathsf{1}+\mathsf{1})) \approx \mathsf{h}(\mathsf{g}(y,\mathsf{1}+\mathsf{1})) \\ &\qquad \to_{\mathsf{ca}} \mathsf{h}(\mathsf{g}(x,v)) \approx \mathsf{h}(\mathsf{g}(y,\mathsf{1}+\mathsf{1}))\ [v = \mathsf{1}+\mathsf{1}] \\ &\qquad \to_{\mathsf{ca}} \mathsf{h}(\mathsf{g}(x,v)) \approx \mathsf{h}(\mathsf{g}(y,z))\ [v = \mathsf{1}+\mathsf{1} \wedge z = \mathsf{1}+\mathsf{1}] \end{aligned}$$

The substitution $\sigma = \{v \mapsto \mathsf{2}, z \mapsto \mathsf{2}\}$ respects the constraint $v = \mathsf{1}+\mathsf{1} \wedge z = \mathsf{1}+\mathsf{1}$ but does not equate $\mathsf{h}(\mathsf{g}(x,v))$ and $\mathsf{h}(\mathsf{g}(y,z))$.

The converse of Corollary 1 also holds, but note that in contrast to TRSs, joinability of critical pairs is not a decidable criterion for terminating LCTRSs, due to the undecidable triviality condition. Moreover, for the converse to hold, it is essential that critical pairs contain the trivial equations ψ in Definition 2.

Example 9. Consider the LCTRS $\mathcal{R}$ consisting of the rules

$$\mathsf{f}(x) \to \mathsf{g}(y) \qquad\qquad \mathsf{g}(y) \to \mathsf{a}\ [y = y]$$

which admits the critical pair $\mathsf{g}(y) \approx \mathsf{g}(y')\ [y = y \wedge y' = y']$ originating from the overlap $\langle \mathsf{f}(x) \to \mathsf{g}(y), \epsilon, \mathsf{f}(x') \to \mathsf{g}(y') \rangle$. This critical pair is joinable as y and y' are restricted to values and thus both sides rewrite to a using the second rule. As $\mathcal{R}$ is also terminating, it is confluent by Corollary 1. If we were to drop ψ in Definition 2, we would obtain the non-joinable critical pair $\mathsf{g}(y) \approx \mathsf{g}(y')$ instead and wrongly conclude non-confluence.

4 Main Results

We start with extending a confluence result of Huet [4] for linear TRSs. Below we write $\to_{\geqslant p}$ to indicate that the position of the contracted redex in the step is below position p.

Definition 6. *A critical pair* $s \approx t\ [\varphi]$ *is* strongly closed *if*

1. $s \approx t\ [\varphi] \stackrel{\sim}{\to}^{*}_{\geqslant 1} \cdot \stackrel{\sim}{\to}^{=}_{\geqslant 2} u \approx v\ [\psi]$ *for some trivial* $u \approx v\ [\psi]$*, and*
2. $s \approx t\ [\varphi] \stackrel{\sim}{\to}^{*}_{\geqslant 2} \cdot \stackrel{\sim}{\to}^{=}_{\geqslant 1} u \approx v\ [\psi]$ *for some trivial* $u \approx v\ [\psi]$*.*

A binary relation $\to$ on terms is *strongly confluent* if $t \to^{*} \cdot {}^{=}\!\!\leftarrow u$ for all terms s, t and u with $t \leftarrow s \to u$. (By symmetry, also $t \to^{=} \cdot {}^{*}\!\!\leftarrow u$ is required.) Strong confluence is a well-known sufficient condition for confluence. Huet [4] proved that linear TRSs are strongly confluent if all critical pairs are strongly closed. Below we extend this result to LCTRSs, using the above definition of strongly closed constrained critical pairs.

Theorem 2. *A linear LCTRS is strongly confluent if all its critical pairs are strongly closed.*

We give full proof details in order to illustrate the complications caused by constrained rewrite rules. The following result from [12] plays an important role.

Lemma 2. *Suppose $s \approx t\ [\varphi] \overset{\sim}{\to}_p u \approx v\ [\psi]$ and $\gamma \vDash \varphi$. If $p \geqslant 1$ then $s\gamma \to u\delta$ and $t\gamma = v\delta$ for some substitution δ with $\delta \vDash \psi$. If $p \geqslant 2$ then $s\gamma = u\delta$ and $t\gamma \to v\delta$ for some substitution δ with $\delta \vDash \psi$.* □

Proof (of Theorem 2). Consider an arbitrary local peak

$$t \leftarrow_{p_1\,|\,\rho_1\,|\,\sigma_1} s \to_{p_2\,|\,\rho_2\,|\,\sigma_2} u$$

with rewrite rules $\rho_1 \colon \ell_1 \to r_1\ [\varphi_1]$ and $\rho_2 \colon \ell_2 \to r_2\ [\varphi_2]$ from $\mathcal{R} \cup \mathcal{R}_{\mathsf{ca}}$. We may assume that ρ_1 and ρ_2 have no variables in common, and consequently $\mathcal{D}\mathsf{om}(\sigma_1) \cap \mathcal{D}\mathsf{om}(\sigma_2) = \varnothing$. We have $s|_{p_1} = \ell_1\sigma_1$, $t = s[r_1\sigma_1]_{p_1}$ and $\sigma_1 \vDash \varphi_1$. Likewise, $s|_{p_2} = \ell_2\sigma_2$, $u = s[r_2\sigma_2]_{p_2}$ and $\sigma_2 \vDash \varphi_2$. If $p_1 \parallel p_2$ then

$$t \to_{p_2\,|\,\rho_2\,|\,\sigma_2} t[r_2\sigma_2]_{p_2} = u[r_1\sigma_1]_{p_1} \leftarrow_{p_1\,|\,\rho_1\,|\,\sigma_1} u$$

Hence both $t \to^{*} \cdot {}^{=}\!\!\leftarrow u$ and $t \to^{=} \cdot {}^{*}\!\!\leftarrow u$. If p_1 and p_2 are not parallel then $p_1 \leqslant p_2$ or $p_2 < p_1$. Without loss of generality, we consider $p_1 \leqslant p_2$. Let $q = p_2 \backslash p_1$. We do a case analysis on whether or not $q \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell_1)$.

- First suppose $q \notin \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell_1)$. Let $q = q_1q_2$ such that $q_1 \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(\ell_1)$ and let x be the variable in ℓ_1 at position q_1. We have $\ell_2\sigma_2 = x\sigma_1|_{q_2}$ and thus $\sigma_1(x) \notin \mathcal{V}\mathsf{al}$. Define the substitution σ_1' as follows:

$$\sigma_1'(y) = \begin{cases} x\sigma_1[r_2\sigma_2]_{q_2} & \text{if } y = x \\ \sigma_1(y) & \text{otherwise} \end{cases}$$

  We show $t \to^{=} s[r_1\sigma_1']_{p_1} \leftarrow u$, which yields $t \to^{*} \cdot {}^{=}\!\!\leftarrow u$ and $t \to^{=} \cdot {}^{*}\!\!\leftarrow u$. Since $\mathcal{R}$ is left-linear, $\ell_1\sigma_1' = \ell_1\sigma_1[x\sigma_1']_{q_1} = \ell_1\sigma_1[x\sigma_1[r_2\sigma_2]_{q_2}]_{q_1} = \ell_1\sigma_1[r_2\sigma_2]_q$ and thus $u = s[r_2\sigma_2]_{p_2} = s[\ell_1\sigma_1[r_2\sigma_2]_q]_{p_1} = s[\ell_1\sigma_1']_{p_1}$. If we can show $\sigma_1' \vDash \rho_1$ then $u \to s[r_1\sigma_1']_{p_1}$. Consider an arbitrary variable $y \in \mathcal{L}\mathcal{V}\mathsf{ar}(\rho_1)$. If $y \neq x$ then $\sigma_1'(y) = \sigma_1(y) \in \mathcal{V}\mathsf{al}$ since $\sigma_1 \vDash \rho_1$. If $y = x$ then $x \in \mathcal{V}\mathsf{ar}(\varphi)$ since $x \in \mathcal{V}\mathsf{ar}(\ell_1)$. However, this contradicts $\sigma_1 \vDash \rho_1$ as $\sigma_1(x) \notin \mathcal{V}\mathsf{al}$. So $\sigma_1'(y) = \sigma_1(y)$ for all $y \in \mathcal{L}\mathcal{V}\mathsf{ar}(\rho_1)$ and thus $\sigma_1' \vDash \rho_1$ is an immediate consequence of $\sigma_1 \vDash \rho_1$. It remains to show $t \to^{=} s[r_1\sigma_1']_{p_1}$. If $x \notin \mathcal{V}\mathsf{ar}(r_1)$ then $r_1\sigma_1' = r_1\sigma_1$ and thus $t = s[r_1\sigma_1']_{p_1}$. If $x \in \mathcal{V}\mathsf{ar}(r_1)$ then there exists a unique position

$q' \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(r_1)$ such that $r_1|_{q'} = x$, due to the right-linearity of $\mathcal{R}$. Hence $r_1\sigma_1' = r_1\sigma_1[x\sigma_1[r_2\sigma_2]_{q_2}]_{q'} = r_1\sigma_1[r_2\sigma_2]_{q'q_2}$. Since $r_1\sigma_1|_{q'q_2} = \ell_2\sigma_2$ we obtain $t = s[r_1\sigma_1]_{p_1} \to_{p_1q'q_2\,|\,\rho_2\,|\,\sigma_2} s[r_1\sigma_1']_{p_1}$ as desired.

- Next suppose $q \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell_1)$. The substitution $\sigma' = \sigma_1 \cup \sigma_2$ satisfies $\ell_1|_q\sigma' = \ell_1|_q\sigma_1 = \ell_2\sigma_2 = \ell_2\sigma'$ and thus is a unifier of $\ell_1|_q$ and ℓ_2. Since $\sigma_1 \vDash \rho_1$ and $\sigma_2 \vDash \rho_2$, $\sigma'(x) \in \mathcal{V}\mathsf{al}$ for all $x \in \mathcal{LV}\mathsf{ar}(\rho_1) \cup \mathcal{LV}\mathsf{ar}(\rho_2)$. Let σ be an mgu of $\ell_1|_q$ and ℓ_2. Since σ is at least as general as σ', $\sigma(x) \in \mathcal{V}\mathsf{al} \cup \mathcal{V}$ for all $x \in \mathcal{LV}\mathsf{ar}(\rho_1) \cup \mathcal{LV}\mathsf{ar}(\rho_2)$. Since $\varphi_1\sigma' = \varphi_1\sigma_1$ and $\varphi_2\sigma' = \varphi_2\sigma_2$ are valid, $\varphi_1\sigma \wedge \varphi_2\sigma$ is satisfiable. Hence conditions 1, 2, 3 and 4 in Definition 2 hold for the triple $\langle \rho_2, q, \rho_1 \rangle$. If condition 5 is *not* fulfilled then $q = \epsilon$ (and thus $p_1 = p_2$), ρ_2 and ρ_1 are variants, and $\mathcal{V}\mathsf{ar}(r_2) \subseteq \mathcal{V}\mathsf{ar}(\ell_2)$ (and thus also $\mathcal{V}\mathsf{ar}(r_1) \subseteq \mathcal{V}\mathsf{ar}(\ell_1)$). Hence $\ell_1\sigma_1 = \ell_2\sigma_2$ and $r_1\sigma_1 = r_2\sigma_2$, and thus $t = u$. In the remaining case condition 5 holds and hence $\langle \rho_2, q, \rho_1 \rangle$ is an overlap. By definition, $\ell_1\sigma[r_2\sigma]_q \approx r_1\sigma\ [\varphi_2\sigma \wedge \varphi_1\sigma \wedge \psi\sigma]$ with

$$\psi = \bigwedge \{x = x \mid x \in \mathcal{EV}\mathsf{ar}(\rho_1) \cup \mathcal{EV}\mathsf{ar}(\rho_2)\}$$

is a critical pair. To simplify the notation, we abbreviate $\ell_1\sigma[r_2\sigma]_q$ to s', $r_1\sigma$ to t', and $\varphi_2\sigma \wedge \varphi_1\sigma \wedge \psi\sigma$ to φ'. Critical pairs are strongly closed by assumption, and thus both

1. $s' \approx t'\ [\varphi']\ \overset{\sim}{\to}{}^{*}_{\geqslant 1} \cdot \overset{\sim}{\to}{}^{=}_{\geqslant 2}\ u \approx v\ [\psi']$ for some trivial $u \approx v\ [\psi']$, and
2. $s' \approx t'\ [\varphi']\ \overset{\sim}{\to}{}^{*}_{\geqslant 2} \cdot \overset{\sim}{\to}{}^{=}_{\geqslant 1}\ u \approx v\ [\psi']$ for some trivial $u \approx v\ [\psi']$.

Let γ be the substitution such that $\sigma\gamma = \sigma'$. We claim that γ respects φ'. So let $x \in \mathcal{V}\mathsf{ar}(\varphi') = \mathcal{V}\mathsf{ar}(\varphi_2\sigma \wedge \varphi_1\sigma \wedge \psi\sigma)$. We have

$$\mathcal{LV}\mathsf{ar}(\rho_1) = \mathcal{V}\mathsf{ar}(\varphi_1) \cup \mathcal{EV}\mathsf{ar}(\rho_1) \qquad \mathcal{LV}\mathsf{ar}(\rho_2) = \mathcal{V}\mathsf{ar}(\varphi_2) \cup \mathcal{EV}\mathsf{ar}(\rho_2)$$

Together with $\mathcal{V}\mathsf{ar}(\psi) = \mathcal{EV}\mathsf{ar}(\rho_1) \cup \mathcal{EV}\mathsf{ar}(\rho_2)$ we obtain

$$\mathcal{LV}\mathsf{ar}(\rho_1) \cup \mathcal{LV}\mathsf{ar}(\rho_2) = \mathcal{V}\mathsf{ar}(\varphi_1) \cup \mathcal{V}\mathsf{ar}(\varphi_2) \cup \mathcal{V}\mathsf{ar}(\psi)$$

Since $\sigma'(x) \in \mathcal{V}\mathsf{al}$ for all $x \in \mathcal{LV}\mathsf{ar}(\rho_1) \cup \mathcal{LV}\mathsf{ar}(\rho_2)$, we obtain $\gamma(x) \in \mathcal{V}\mathsf{al}$ for all $x \in \mathcal{V}\mathsf{ar}(\varphi')$ and thus $\gamma \vDash \varphi'$. At this point repeated applications of Lemma 2 to the constrained rewrite sequence in item 1 yields a substitution δ respecting ψ' such that $s'\gamma \to^* u\delta$ and $t'\gamma = v\delta$. Since $u \approx v\ [\psi']$ is trivial, $u\delta = v\delta$ and hence $s'\gamma \to^* \cdot {}^{=}\!\leftarrow t'\gamma$. Likewise, $s'\gamma \to^{=} \cdot {}^{*}\!\leftarrow t'\gamma$ is obtained from item 2. We have

$$s'\gamma = (\ell_1\sigma[r_2\sigma]_q)\gamma = \ell_1\sigma'[r_2\sigma']_q = \ell_1\sigma_1[r_2\sigma_2]_q \qquad t'\gamma = r_1\sigma' = r_1\sigma_1$$

Moreover, $t = s[r_1\sigma_1]_{p_1} = s[t'\gamma]_{p_1}$ and $u = s[\ell_1\sigma_1[r_2\sigma_2]_q]_{p_1} = s[s'\gamma]_{p_1}$. Since rewriting is closed under contexts, we obtain $u \to^* \cdot {}^{=}\!\leftarrow t$ and $u \to^{=} \cdot {}^{*}\!\leftarrow t$. This completes the proof. □

Example 10. Consider the LCTRS $\mathcal{R}$ of Example 1 and its critical pairs in Example 4. The critical pair

$$x \approx \mathsf{max}(y, x)\ [x \geq y]$$

is not trivial, so Theorem 1 is not applicable and the rule $\mathsf{max}(x,y) \to \mathsf{max}(y,x)$ precludes the use of Corollary 1 to infer confluence. We do have

$$x \approx \mathsf{max}(y,x)\ [x \geq y] \xrightarrow{\geqslant 2} x \approx x\ [x \geq y]$$

by applying the rule $\mathsf{max}(x,y) \to y\ [y \geq x]$ and the resulting constrained equation $x \approx x\ [x \geq y]$ is obviously trivial. The same reasoning applies to the critical pair $y \approx \mathsf{max}(y,x)\ [y \geq x]$. The first critical pair $x \approx y\ [x \geq y \wedge y \geq x]$ in Example 4 is trivial since any (value) substitution satisfying its constraint $x \geq y \wedge y \geq x$ equates x and y. By symmetry, all critical pairs of $\mathcal{R}$ are strongly closed. Since $\mathcal{R}$ is linear, confluence follows from Theorem 2.

The second main result is the extension of Huet's parallel closedness condition on critical pairs in left-linear TRSs [4] to LCTRSs. To this end, we first define parallel rewriting for LCTRSs.

Definition 7. *Let $\mathcal{R}$ be an LCTRS. The relation $\mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}}$ is defined on terms inductively as follows:*

1. *$x \mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}} x$ for all variables x,*
2. *$f(s_1,\ldots,s_n) \mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}} f(t_1,\ldots,t_n)$ if $s_i \mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}} t_i$ for all $1 \leqslant i \leqslant n$,*
3. *$\ell\sigma \mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}} r\sigma$ with $\ell \to r\ [\varphi] \in \mathcal{R}$ and $\sigma \vDash \ell \to r\ [\varphi]$,*
4. *$f(v_1,\ldots,v_n) \mathrel{+\!\!\!\!\rightarrow} v$ with $f \in \mathcal{F}_{\mathsf{th}} \setminus \mathcal{V}\mathsf{al}$, $v_1,\ldots,v_n \in \mathcal{V}\mathsf{al}$ and $v = [\![f(v_1,\ldots,v_n)]\!]$.*

We write $\mathrel{+\!\!\!\!\rightarrow}_{\geqslant p}$ to indicate that all positions of contracted redexes in the parallel step are below p. In the next definition we add constraints to parallel rewriting.

Definition 8. *Let $\mathcal{R}$ be an LCTRS. The relation $\mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}}$ is defined on constrained terms inductively as follows:*

1. *$x\ [\varphi] \mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}} x\ [\varphi]$ for all variables x,*
2. *$f(s_1,\ldots,s_n)\ [\varphi] \mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}} f(t_1,\ldots,t_n)\ [\varphi \wedge \psi]$ if $s_i\ [\varphi] \mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}} t_i\ [\varphi \wedge \psi_i]$ for all $1 \leqslant i \leqslant n$ and $\psi = \psi_1 \wedge \cdots \wedge \psi_n$,*
3. *$\ell\sigma\ [\varphi] \mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}} r\sigma\ [\varphi]$ with $\rho\colon \ell \to r\ [\omega] \in \mathcal{R}$, $\sigma(x) \in \mathcal{V}\mathsf{al} \cup \mathcal{V}\mathsf{ar}(\varphi)$ for all $x \in \mathcal{LV}\mathsf{ar}(\rho)$, φ is satisfiable and $\varphi \Rightarrow \omega\sigma$ is valid,*
4. *$f(v_1,\ldots,v_n)\ [\varphi] \mathrel{+\!\!\!\!\rightarrow} v\ [\varphi \wedge v = f(v_1,\ldots,v_n)]$ with $v_1,\ldots,v_n \in \mathcal{V}\mathsf{al} \cup \mathcal{V}\mathsf{ar}(\varphi)$, $f \in \mathcal{F}_{\mathsf{th}} \setminus \mathcal{V}\mathsf{al}$ and v is a fresh variable.*

Here we assume that different applications to case 4 result in different fresh variables. The constraint ψ in case 2 collects the assignments introduced in earlier applications of case 4. (If there are none, $\psi = \mathsf{true}$ is omitted.) The same holds for $\psi_1,\ldots,\psi_n$. We write $\mathrel{\widetilde{+\!\!\!\!\rightarrow}}$ for the relation $\sim \cdot \mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}} \cdot \sim$.

In light of the earlier developments, the following definition is the obvious adaptation of parallel closedness for LCTRSs.

Definition 9. *A critical pair* $s \approx t\ [\varphi]$ *is* parallel closed *if*

$$s \approx t\ [\varphi] \mathrel{\widetilde{+\!\!\!\!\!\!\to}}_{\geqslant 1} u \approx v\ [\psi]$$

for some trivial $u \approx v\ [\psi]$.

Note that the right-hand side t of the constrained equation $s \approx t\ [\varphi]$ may change due to the equivalence relation $\sim$, cf. the statement of Lemma 2.

Theorem 3. *A left-linear LCTRS is confluent if its critical pairs are parallel closed.*

To prove this result, we adapted the formalized proof presented in [10] to the constrained setting. The required changes are very similar to the ones in the proof of Theorem 2.

Example 11. Consider the LCTRS $\mathcal{R}$ with rules

$$\begin{aligned} \mathsf{f}(x,y) &\to \mathsf{g}(\mathsf{a}, y+y)\ [y \geq x \wedge y = 1] & \mathsf{a} &\to \mathsf{b} \\ \mathsf{h}(\mathsf{f}(x,y)) &\to \mathsf{h}(\mathsf{g}(\mathsf{b},2))\ [x \geq y] & \mathsf{g}(x,y) &\to \mathsf{g}(y,x) \end{aligned}$$

The single critical pair $\mathsf{h}(\mathsf{g}(\mathsf{a}, y+y)) \approx \mathsf{h}(\mathsf{g}(\mathsf{b},2))\ [y \geq x \wedge y = 1 \wedge x \geq y]$ is parallel closed:

$$\begin{aligned} &\mathsf{h}(\mathsf{g}(\mathsf{a}, y+y)) \approx \mathsf{h}(\mathsf{g}(\mathsf{b},2))\ [y \geq x \wedge y = 1 \wedge x \geq y] \\ &\qquad \mathrel{+\!\!\!\!\!\!\to}_{\geqslant 1} \mathsf{h}(\mathsf{g}(\mathsf{b}, z)) \approx \mathsf{h}(\mathsf{g}(\mathsf{b},2))\ [y \geq x \wedge y = 1 \wedge x \geq y \wedge z = y+y] \end{aligned}$$

and the obtained equation is trivial. Hence $\mathcal{R}$ is confluent by Theorem 3. Note that the earlier confluence criteria do not apply.

We also consider the extension of Huet's result by Toyama [11], which has a less restricted joinability condition on critical pairs stemming from overlapping rules at the root position. Such critical pairs are called *overlays* whereas critical pairs originating from overlaps $\langle \rho_1, p, \rho_2 \rangle$ with $p > \epsilon$ are called *inner* critical pairs.

Definition 10. *An LCTRS* $\mathcal{R}$ *is almost parallel-closed if every inner critical pair is parallel closed and every overlay* $s \approx t\ [\varphi]$ *satisfies*

$$s \approx t\ [\varphi] \mathrel{\widetilde{+\!\!\!\!\!\!\to}}_{\geqslant 1} \cdot \mathrel{\tilde{\to}}^{*}_{\geqslant 2} u \approx v\ [\psi]$$

for some trivial $u \approx v\ [\psi]$.

Theorem 4. *Left-linear almost parallel-closed LCTRSs are confluent.*

Again, the formalized proof of the corresponding result for plain TRSs in [10] can be adapted to the constrained setting.

Example 12. Consider the following variation of the LCTRS $\mathcal{R}$ in Example 11:

$$\begin{aligned} \mathsf{f}(x,y) &\to \mathsf{g}(\mathsf{a}, y+y)\ [y \geq x \wedge y = 1] & \mathsf{a} &\to \mathsf{b} \\ \mathsf{f}(x,y) &\to \mathsf{g}(\mathsf{b},2)\ [x \geq y] & \mathsf{g}(x,y) &\to \mathsf{g}(y,x) \end{aligned}$$

The overlay $\mathsf{g}(\mathsf{b},2) \approx \mathsf{g}(\mathsf{a}, y+y)\ [x \geq y \wedge y \geq x \wedge y = 1]$ is not parallel closed but one readily confirms that the condition in Definition 10 applies.

5 Automation

As it is very inconvenient and tedious to test by hand if an LCTRS satisfies one of the confluence criteria presented in the preceding sections, we provide an implementation. The natural choice would be to extend the existing tool Ctrl [9] because it is currently the only tool capable of analyzing confluence of LCTRSs. However, Ctrl is not actively maintained and not very well documented, so we decided to develop a new tool for the analysis of LCTRSs. Our tool is called crest (constrained rewriting software). It is written in Haskell, based on the Haskell `term-rewriting`[3] library and allows the logics `QF_LIA`, `QF_NIA`, `QF_LRA`.

The input format of crest is described on its website.[4] After parsing the input, crest checks that the resulting LCTRS is well-typed. Missing sort information is inferred. Next it is checked concurrently whether one of the implemented confluence criteria applies. crest supports (weak) orthogonality, strong closedness and (almost) parallel closedness. The tool outputs the computed critical pairs and a "proof" describing how these are closed, based on the first criterion that reports a YES result. Below we describe some of the challenges that one faces when automating the confluence criteria presented in the preceding sections.

First of all, how can we determine whether a constrained critical pair or more generally a constrained equation $s \approx t\ [\varphi]$ is trivial? The following result explains how this can be solved by an SMT solver.

Definition 11. *Given a constrained equation $s \approx t\ [\varphi]$, the formula $T(s,t,\varphi)$ is inductively defined as follows:*

$$T(s,t,\varphi) = \begin{cases} \mathsf{true} & \text{if } s = t \\ s = t & \text{if } s,t \in \mathcal{V}\mathsf{al} \cup \mathcal{V}\mathsf{ar}(\varphi) \\ \bigwedge_{i=1}^{n} T(s_i,t_i,\varphi) & \text{if } s = f(s_1,\dots,s_n) \text{ and } t = f(t_1,\dots,t_n) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Lemma 3. *A constrained equation $s \approx t\ [\varphi]$ is trivial if and only if the formula $\varphi \implies T(s,t,\varphi)$ is valid.*

Proof. First suppose $\varphi \implies T(s,t,\varphi)$ is valid. Let σ be a substitution with $\sigma \vDash \varphi$. Since $\sigma(x) \in \mathcal{V}\mathsf{al}$ for all $x \in \mathcal{V}\mathsf{ar}(\varphi)$, we can apply σ to the formula $\varphi \implies T(s,t,\varphi)$. We obtain $[\![\varphi\sigma]\!] = \top$ from $\sigma \vDash \varphi$. Hence also $[\![T(s,t,\varphi)\sigma]\!] = \top$. Since $T(s,t,\varphi)$ is a conjunction, the final case in the definition of $T(s,t,\varphi)$ is not used. Hence $\mathcal{P}\mathsf{os}(s) = \mathcal{P}\mathsf{os}(t)$, $s(p) = t(p)$ for all internal positions p in s and t, and $s|_p\sigma = t|_p\sigma$ for all leaf positions p in s and t. Consequently, $s\sigma = t\sigma$. This concludes the triviality proof of $s \approx t\ [\varphi]$.

For the only if direction, suppose $s \approx t\ [\varphi]$ is trivial. Note that the variables appearing in the formula $\varphi \implies T(s,t,\varphi)$ are those of φ. Let σ be an arbitrary

[3] https://hackage.haskell.org/package/term-rewriting-0.4.0.2.

[4] http://cl-informatik.uibk.ac.at/software/crest/.

assignment such that $[\![\varphi\sigma]\!] = \top$. We need to show $[\![T(s,t,\varphi)\sigma]\!] = \top$. We can view σ as a substitution with $\sigma(x) \in \mathcal{V}\mathsf{al}$ for all $x \in \mathcal{V}\mathsf{ar}(\varphi)$. We have $\sigma \vDash \varphi$ and thus $s\sigma = t\sigma$ by the triviality of $s \approx t\ [\varphi]$. Hence $T(s,t,\varphi)$ is a conjunction of equations between values and variables in φ, which are turned into identities by σ. Hence $[\![T(s,t,\varphi)\sigma]\!] = \top$ as desired. □

The second challenge is how to implement rewriting on constrained equations in particular, how to deal with the equivalence relation $\sim$ defined in Definition 3.

Example 13. The LCTRS $\mathcal{R}$

$$\mathsf{f}(x) \to z\ [z = 3] \qquad \mathsf{g}(\mathsf{f}(x)) \to \mathsf{a} \qquad \mathsf{g}(3) \to \mathsf{a}$$

over the integers admits two critical pairs:

$$z \approx z'\ [z = 3 \wedge z' = 3] \qquad \mathsf{g}(z) \approx \mathsf{a}\ [z = 3]$$

The first one is trivial, but to join the second one, an initial equivalence step is required:

$$\mathsf{g}(z) \approx \mathsf{a}\ [z = 3] \sim \mathsf{g}(3) \approx \mathsf{a}\ [z = 3] \to \mathsf{a} \approx \mathsf{a}\ [z = 3]$$

The transformation introduced below avoids having to look for an initial equivalence step before a rule becomes applicable.

Definition 12. *Let $\mathcal{R}$ be an LCTRS. Given a term $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$, we replace values in t by fresh variables and return the modified term together with the constraint that collects the bindings:*

$$\mathsf{tf}(t) = \begin{cases} (t, \mathsf{true}) & \textit{if } t \in \mathcal{V} \\ (z, z = t) & \textit{if } t \in \mathcal{V}\mathsf{al} \textit{ and } z \textit{ is a fresh variable} \\ (f(s_1,\dots,s_n), \varphi_1 \wedge \cdots \wedge \varphi_n) & \textit{if } t = f(t_1,\dots,t_n) \textit{ and } \mathsf{tf}(t_i) = (s_i,\varphi_i) \end{cases}$$

Applying the transformation tf *to the left-hand sides of the rules in $\mathcal{R}$ produces*

$$\mathsf{tf}(\mathcal{R}) = \{\ell' \to r\ [\varphi \wedge \psi] \mid \ell \to r\ [\varphi] \in \mathcal{R} \textit{ and } \mathsf{tf}(\ell) = (\ell',\psi)\}$$

Example 14. Applying the transformation tf to the LCTRS $\mathcal{R}$ of Example 13 produces the rules

$$\mathsf{f}(x) \to z\ [z = 3] \qquad \mathsf{g}(\mathsf{f}(x)) \to \mathsf{a} \qquad \mathsf{g}(z) \to \mathsf{a}\ [z = 3]$$

The critical pair $\mathsf{g}(z) \approx \mathsf{a}\ [z = 3]$ can now be joined by an application of the modified third rule. Note that the modified rule does not overlap with the second rule because z may not be instantiated with $\mathsf{f}(x)$. Hence the modified LCTRS $\mathsf{tf}(\mathcal{R})$ is strongly closed and, because it is linear, also confluent.

In the following we show the correctness of the transformation. In particular we prove that the initial rewrite relation is preserved.

Table 1. Specific experimental results.

	result	method	time (in ms)
[12, Example 23]	Timeout	–	10017.70
[12, Example 23] corrected	YES	strongly closed	103.71
Example 6	YES	orthogonal	34.35
[8, Example 3]	YES	weakly orthogonal	50.87
Example 1	YES	strongly closed	115.33
[10, Example 1]	YES	strongly closed	3806.84
Example 11	YES	parallel closed	38.42
Example 12	YES	almost parallel closed	130.36

Lemma 4. *The relations* $\to_{\mathcal{R}}$ *and* $\to_{\mathsf{tf}(\mathcal{R})}$ *coincide on unconstrained terms.*

Proof. Consider $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. Since the transformation tf does not affect calculation steps, it suffices to consider rule steps. First assume $s = C[\ell\sigma] \to_{\mathsf{ru}} C[r\sigma] = t$ by applying the rule $\ell \to r\ [\varphi] \in \mathcal{R}$ and let $\ell' \to r\ [\varphi'] \in \mathsf{tf}(\mathcal{R})$ be its transformation. So $\mathsf{tf}(\ell) = (\ell', \psi)$ and $\varphi' = \varphi \wedge \psi$. Define the substitution

$$\sigma' = \{\ell'|_p \mapsto \ell|_p \mid (\ell', \psi) = \mathsf{tf}(\ell), p \in \mathcal{P}\mathsf{os}(\ell) \text{ and } \ell|_p \in \mathcal{V}\mathsf{al}\}$$

and let $\tau = \sigma \cup \sigma'$. Since $\mathcal{D}\mathsf{om}(\sigma) \cap \mathcal{D}\mathsf{om}(\sigma') = \varnothing$ by construction, τ is well-defined. From $\sigma \vDash \ell \to r\ [\varphi]$ and $\sigma' \vDash \psi$ we immediately obtain $\tau \vDash \ell' \to r\ [\varphi']$, which yields $s = C[\ell'\tau] \to_{\mathsf{ru}} C[r\tau] = t$ in $\mathsf{tf}(\mathcal{R})$.

For the other direction consider $s = C[\ell'\sigma] \to_{\mathsf{ru}} C[r'\sigma] = t$ by applying the rule $\ell' \to r'\ [\varphi'] \in \mathsf{tf}(\mathcal{R})$. The difference between ℓ' and its originating left-hand side ℓ in $\mathcal{R}$ is that value positions in ℓ are occupied by fresh variables in ℓ'. Because σ' respects $\varphi' = \varphi \wedge \psi$, σ' substitutes the required values at these positions in ℓ. As $\sigma \vDash \ell' \to r'\ [\varphi']$, there exists a rule $\ell \to r\ [\varphi]$ which is respected by σ and thus $s = C[\ell\sigma] \to_{\mathsf{ru}} C[r\sigma] = t$ in $\mathcal{R}$. □

As the transformation is used in the implementation and rewriting on constrained terms plays a key role, the following result is needed. The proof is similar to the first half of the proof of Lemma 4 and omitted.

Lemma 5. *The inclusion* $\to_{\mathcal{R}} \subseteq \to_{\mathsf{tf}(\mathcal{R})}$ *holds on constrained terms.*

6 Experimental Results

In order to evaluate our tool we performed some experiments. As there is no official database of interesting confluence problems for LCTRSs, we collected several LCTRSs from the literature and the repository of Ctrl. The problem files in the latter that contain an equivalence problem of two functions for rewriting induction were split into two separate files. The experiments were performed on an AMD Ryzen 7 PRO 4750U CPU with a base clock speed of 1.7 GHz, 8

Table 2. Comparison between confluence criteria implemented in crest.

	O	W	S	P	A
orthogonality (O)	74	74	11	74	74
weak orthogonality (W)		78	13	78	78
strongly closed (S)			20	16	20
parallel closed (P)				83	83
almost parallel closed (A)					89

cores and 32 GB of RAM. The full set of benchmarks consists of 127 problems of which crest can prove 90 confluent, 11 result in MAYBE and 26 in a timeout. With a timeout of 5 s crest needs 141.09 s to analyze the set of benchmarks. We have tested the implementation with 3 well-known SMT solvers: Z3, Yices and CVC5. Among those Z3 gives the best performance regarding time and the handling of non-linear arithmetic. Hence we use Z3 as the default SMT solver in our implementation. In Table 1 we list some interesting systems from this paper and the relevant literature. Full details are available from the website of crest. We choose 5 as the maximum number of steps in the $\to^*$ parts of the strongly closed and almost parallel closed criteria.

From Table 2 the relative power of each implemented confluence criterion on our benchmark can be inferred, i.e., it depicts how many of the 127 problems both methods can prove confluent. This illustrates that the relative applicability in theory (e.g., weakly orthogonal LCTRSs are parallel closed), is preserved in our implementation. We conclude this section with an interesting observation discovered by crest when testing [12, Example 23].

We also tested the applicability of Corollary 1, using the tool Ctrl as a black box for proving termination. Of the 127 problems, Ctrl claims 102 to be terminating and 67 of those can be shown locally confluent by crest, where we limit the number of steps in the joining sequence to 100. It is interesting to note that all of these problems are orthogonal, and so proving termination and finding a joining sequence is not necessary to conclude confluence, on the current set of problems. Of the remaining 35 problems, crest can show confluence of 5 of these by almost parallel closedness.

Example 15. The LCTRS $\mathcal{R}$ is obtained by completing a system consisting of four constrained equations:

1. $\mathsf{f}(x, y) \to \mathsf{f}(z, y) + \mathsf{1}\ [x \geq \mathsf{1} \land z = x - \mathsf{1}]$
2. $\mathsf{f}(x, \mathsf{0}) \to \mathsf{g}(\mathsf{1}, x)\ [x \leq \mathsf{1}]$
3. $\mathsf{g}(\mathsf{0}, y) \to y\ [x \leq \mathsf{0}]$ 5. $\mathsf{h}(x) \to \mathsf{g}(\mathsf{1}, x) + \mathsf{1}\ [x \leq \mathsf{1}]$
4. $\mathsf{g}(\mathsf{1}, \mathsf{1}) \to \mathsf{g}(\mathsf{1}, \mathsf{0}) + \mathsf{1}$ 6. $\mathsf{h}(x) \to \mathsf{f}(x - \mathsf{1}, \mathsf{0}) + \mathsf{2}\ [x \geq \mathsf{1}]$

Calling crest on $\mathcal{R}$ results in a timeout. As a matter of fact, the LCTRS is not confluent because the critical pair

$$\mathsf{g}(\mathsf{1}, x) + \mathsf{1} \approx \mathsf{f}(x - \mathsf{1}, \mathsf{0}) + \mathsf{2}\ [x \leq \mathsf{1} \land x \geq \mathsf{1}]$$

between rules 5 and 6 is not joinable. Inspecting the steps in [12, Example 23] reveals some incorrect applications of the inference rules of constrained completion, which causes rule 6 to be wrong. Replacing it with the correct rule

$$6'.\quad \mathsf{h}(x) \to (\mathsf{f}(z, \mathsf{0}) + \mathsf{1}) + \mathsf{1}\ [x > \mathsf{1} \wedge z = x - \mathsf{1}]$$

causes crest to report confluence by strong closedness.

7 Concluding Remarks

In this paper we presented new confluence criteria for LCTRSs as well as a new tool in which these criteria have been implemented. We clarified the subtleties that arise when analyzing joinability of critical pairs in LCTRSs and reported experimental results.

For plain rewrite systems many more confluence criteria are known and implemented in powerful tools that compete in the yearly Confluence Competition (CoCo).[5] In the near future we will investigate which of these can be lifted to LCTRSs. We will also advance the creation of a competition category on confluence of LCTRSs in CoCo.

Our tool crest has currently no support for termination. Implementing termination techniques in crest is of clear interest. The starting point here are the methods reported in [6,7,12]. Many LCTRSs coming from applications are actually non-confluent.[6] So developing more powerful techniques for LCTRSs is on our agenda as well.

Acknowledgments. We thank Fabian Mitterwallner for valuable discussions on the presented topics and our Haskell implementation. The detailed comments by the reviewers improved the presentation. Cynthia Kop and Deivid Vale kindly provided us with instructions and a working implementation of Ctrl.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998). https://doi.org/10.1017/CBO9781139172752
2. Ciobâcă, Ş, Lucanu, D.: A coinductive approach to proving reachability properties in logically constrained term rewriting systems. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 295–311. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_20
3. Fuhs, C., Kop, C., Nishida, N.: Verifying procedural programs via constrained rewriting induction. ACM Trans. Comput. Log. **18**(2), 14:1–14:50 (2017). https://doi.org/10.1145/3060143
4. Huet, G.: Confluent reductions: abstract properties and applications to term rewriting systems. J. ACM **27**(4), 797–821 (1980). https://doi.org/10.1145/322217.322230

[5] http://project-coco.uibk.ac.at/.

[6] Naoki Nishida, personal communication (February 2023).

5. Kojima, M., Nishida, N.: From starvation freedom to all-path reachability problems in constrained rewriting. In: Hanus, M., Inclezan, D. (eds.) PADL 2023. LNCS, vol. 13880, pp. 161–179. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-24841-2_11
6. Kop, C.: Termination of LCTRSs. In: Proceedings of the 13th International Workshop on Termination, pp. 59–63 (2013)
7. Kop, C.: Termination of LCTRSs. CoRR abs/1601.03206 (2016). https://doi.org/10.48550/ARXIV.1601.03206
8. Kop, C., Nishida, N.: Term rewriting with logical constraints. In: Fontaine, P., Ringeissen, C., Schmidt, R.A. (eds.) FroCoS 2013. LNCS (LNAI), vol. 8152, pp. 343–358. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40885-4_24
9. Kop, C., Nishida, N.: Constrained term rewriting tool. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 549–557. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_38
10. Nagele, J., Middeldorp, A.: Certification of classical confluence results for left-linear term rewrite systems. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 290–306. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_18
11. Toyama, Y.: Commutativity of term rewriting systems. In: Fuchi, K., Kott, L. (eds.) Programming of Future Generation Computers II, pp. 393–407. North-Holland (1988)
12. Winkler, S., Middeldorp, A.: Completion for logically constrained rewriting. In: Kirchner, H. (ed.) Proceedings of the 3rd International Conference on Formal Structures for Computation and Deduction. Leibniz International Proceedings in Informatics, vol. 108, pp. 30:1–30:18 (2018). https://doi.org/10.4230/LIPIcs.FSCD.2018.30
13. Winkler, S., Moser, G.: Runtime complexity analysis of logically constrained rewriting. In: LOPSTR 2020. LNCS, vol. 12561, pp. 37–55. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-68446-4_2

Towards a Verified Tableau Prover for a Quantifier-Free Fragment of Set Theory

Lukas Stevens(✉)

Technical University of Munich, Boltzmannstr. 3, 85748 Garching, Germany
lukas.stevens@in.tum.de

Abstract. Using Isabelle/HOL, we verify the state-of-the-art decision procedure for multi-level syllogistic with singleton (**MLSS** for short), which is a quantifier-free fragment of set theory. We formalise its syntax and semantics as well as a sound and complete tableau calculus for it. We also provide an executable specification of a decision procedure that exhaustively applies the rules of the calculus and prove its termination. Furthermore, we extend the calculus with a lightweight type system that paves the way for an integration of the procedure into Isabelle/HOL.

Keywords: Decision procedures · Semantic tableaux · Interactive theorem proving · Set theory

1 Introduction

In Isabelle/HOL, there are specialised procedures for dealing with e.g. natural numbers, linear arithmetic, and metric spaces. Some of these procedures have been verified in Isabelle/HOL, such as a procedure for Presburger arithmetic [12] that was later extended to mixed real-integer arithmetic [11]. This procedure, though, uses reflection to work on goals in Isabelle/HOL, which, during execution, either sacrifices speed by going through the simplifier or requires trusting the code generator. More recently, Stevens and Nipkow [25] presented a verified decision procedure for orders that produces certificates. This approach offers efficient execution by using generated code as well as soundness because the certificates are replayed through Isabelle's inference kernel.

This paper focuses on another ubiquitous structure in mathematics, namely sets. To the best of our knowledge, we present the first formally verified decision procedure for (a fragment of) set theory. In particular, we consider a quantifier-free fragment which Cantone and Zarba [9] call multi-level syllogistic with singleton (**MLSS**). The fragment includes the usual set operations of union, intersection, difference, membership, equality and, in addition, it allows the construction of singleton sets.

Since **MLSS** admits a tableau calculus, generating certificates will be straightforward. Like with the aforementioned order solver, this paves the way

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 491–508, 2023.
https://doi.org/10.1007/978-3-031-38499-8_28

for an integration of the decision procedure into Isabelle, adding to its growing body of verified decision procedures.

1.1 Contributions

We present a formalisation in Isabelle/HOL of a tableau calculus for **MLSS** due to Cantone and Zarba [9] [7, Chapter 14]. We prove soundness and completeness of the calculus and give an abstract specification of a decision procedure that exhaustively applies the rules of the calculus. To obtain total correctness of the procedure, we prove its termination. Additionally, we naively refine the abstract to an executable specification from which we can generate code. The formalisation initially follows the paper but offers a more thorough account of some important details:

- We deliver the omitted proof of Lemma 2 in the paper [9], a key building block for the completeness proof of the calculus.
- The formal proof of completeness reveals that the calculus lacks a rule for eliminating double negation.
- We derive an explicit upper bound for the number of formulas in a tableau branch.

In the context of Isabelle/HOL, there is one crucial aspect that requires us to modify the calculus in the paper: the calculus works under the assumption that every variable is a set; however, this is not the case in Isabelle/HOL, e.g. consider the expression `n ∈ A` where `n` is a natural number. We call these variables urelements. To deal with them, we extend the calculus with a lightweight type system and a verified inference algorithm that identifies the urelements.

The modification of the calculus required non-trivial changes to the completeness proof. Here, the formalisation was instrumental because Isabelle immediately revealed which proofs had been broken. This illustrates the usefulness of ITPs for developing logic calculi: they allow us to confidently make modifications without compromising correctness.

All in all, the formalisation amounts to over 6000 lines of theory. It is part of the *Archive of Formal Proofs* (AFP) [24]. The entry provides an overview theory MLSS_Proc_All.thy that highlights the (mostly syntactic) differences between paper and formalisation and references the constants and theorems that are introduced in this paper.

1.2 Related Work

Since the literature on decidable fragments of set theory is vast, we only focus on **MLSS** here. Ferro et al. [14] were the first to show the decidability of the fragment. Subsequent work [6] found the decision problem to be **NP**-complete. To obtain a practical decision procedure, Cantone [4] proposed a tableau calculus, which was later improved by Beckert and Hartmer [1]. Both of these procedures construct a model during execution that guides the proof search. Beckert and

Hartmer also cover an extension of the calculus with uninterpreted functions, which Cantone and Zarba [10] later revisited while avoiding the construction of a model during execution. In this paper, we consider a version of the latter procedure due to Cantone and Zarba [9] that is specialised to **MLSS** and where the branching rules of the calculus are set up to guarantee the mutual exclusivity of the branches. Later extensions of the calculus added certain interpreted functions, such as monotone functions [8] and the inverse of a function [5]. The latter extension notably includes the Cartesian product. Those extensions, though, did not improve upon the tableau calculus for **MLSS**.

There is a large body of work at the intersection of ITPs and tableau methods, but to keep with this paper's theme we only consider formalisations of correctness here. For first-order logic, there are abstract completeness proofs using the *Beth-Hintikka style* of possibly infinite derivation trees [3] as well as the *Henkin style* of maximally consistent sets [17]. Both are abstract enough to be instantiated with a wide range of concrete calculi. A more concrete formalisation [19] verifies a sequent calculus for first-order logic whose completeness proof is via a translation to semantic tableau.

Beyond completeness, we target decidability, which is more attainable for propositional logic. There is a verified tableau calculus for the modal logic S5 [2] in Lean and one for hybrid logic [18] in Isabelle/HOL. Both of these do not prove termination but there is a formalisation of a tableau calculus for the temporal logic CTL in Coq [13] that does.

1.3 Notation

Isabelle/HOL [21] conforms to everyday mathematical notation for the most part. We establish notation and in particular some essential data types together with their primitive operations that are specific to Isabelle/HOL.

We write `t :: 'a` to specify that the term `t` has the type `'a` and `'a ⇒ 'b` for the space of total functions from type `'a` to type `'b`.

Sets with elements of type `'a` have the type `'a set`. The cardinality of a set `A` is denoted by `|A|` and the image of `A` under `f` by `` f ` A ``.

We use `'a list` to describe the type of lists, which are constructed using the empty list `[]` constructor or the infix cons constructor `#`, and are appended with the infix operator `@`. The function `set` converts a list into a set.

We remark that $\longleftrightarrow$ is equivalent to $=$ on the type of Booleans `bool` and $\equiv$ is definitional equality of the meta-logic of Isabelle/HOL, which is called Isabelle/Pure. Meta-implication is denoted by $\Longrightarrow$ and a chain of implications $\mathtt{A}_1 \Longrightarrow \cdots \Longrightarrow \mathtt{A}_\mathtt{k} \Longrightarrow \mathtt{C}$ can be abbreviated by $[\![\ \mathtt{A}_1;\ldots;\mathtt{A}_\mathtt{k}\ ]\!] \Longrightarrow \mathtt{C}$.

2 Syntax and Semantics of MLSS

2.1 Syntax

At the heart of **MLSS**, we have the type of set terms, which is the disjoint union of the empty set and variables as well as the operations union, intersection, difference, and the singleton set represented by the constructor `Single`. We keep the type of variables abstract by making it a parameter of the set term data type. The only restriction on the type of variables is that it needs to be infinite. Isabelle/HOL's data type package automatically defines a function that gives us the set of variables in a set term, which we name `vars`. In what follows, we will overload the function `vars` to also work on set atoms, formulas, and branches.

```
datatype (vars: 'a) pset_term =
    ∅ | Var 'a | Single ('a pset_term)
  | 'a pset_term ⊔_s 'a pset_term
  | 'a pset_term ⊓_s 'a pset_term
  | 'a pset_term −_s 'a pset_term
```

We can combine two set terms to form a set atom by using the membership or the equality operator.

```
datatype (vars: 'a) pset_atom =
    'a pset_term ∈_s 'a pset_term
  | 'a pset_term =_s 'a pset_term
```

With the above operators we can also represent the subset operator $\sqsubseteq_s$ and enumerate finite sets: $\texttt{s} \sqsubseteq_s \texttt{t}$ is equivalent to $\texttt{s} \sqcup_s \texttt{t} =_s \texttt{t}$ and a finite set of elements $\{\texttt{t}_1,\ldots,\texttt{t}_\texttt{k}\}$ can be expressed by $\texttt{Single t}_1 \sqcup_s \ldots \sqcup_s \texttt{Single t}_k$.

We use the propositional fragment of formulas due to Nipkow [20] with set atoms as propositional atoms to form the quantifier-free fragment **MLSS** of set theory.

```
datatype (atoms: 'a) fm =
    A 'a
  | ¬ ('a fm)
  | 'a fm ∧ 'a fm
  | 'a fm ∨ 'a fm
```

```
type_synonym 'a pset_fm = 'a pset_atom fm
```

We will often drop the atom constructor `A` to reduce clutter. Additionally, we use $\texttt{s} \notin_s \texttt{t}$ and $\texttt{s} \neq_s \texttt{t}$ to denote $\neg\,\texttt{A (s} \in_s \texttt{t)}$ and $\neg\,\texttt{A (s} =_s \texttt{t)}$, respectively.

Similarly to `vars`, we get the function `atoms :: 'a fm ⇒ 'a set` for free that retrieves all set atoms in a formula. We combine these functions to extract all the variables occurring in a set formula.

```
definition vars φ ≡ ⋃(vars ` atoms φ)
```

Likewise, we fix the constant `subterms :: 'b ⇒ 'a pset_term set` that is polymorphic in its argument type `'b`. We overload this constant to return the set terms that are subterms of a set term, set atom, or formula, respectively. Lastly, we introduce the function `subfms :: 'a fm ⇒ 'a fm set` that computes the subformulas of a formula. The functions `subterms` and `subfms` are implemented in the expected way.

2.2 Semantics

The original paper [9] bases the semantics of **MLSS** on the von Neumann hierarchy of sets $\mathcal{V}$. We instead use the hierarchy of *hereditarily finite sets* (HF sets) which fulfil all the same axioms as $\mathcal{V}$ – that is, the axioms of ZF – except for the axiom of infinity. In particular, the membership relation is well-founded. The HF sets, as we will see, are sufficient to construct a model for any satisfiable **MLSS** formula. In contrast to $\mathcal{V}$, the HF sets are directly representable in Isabelle/HOL, and indeed, an AFP entry [23] formalises them. The entry defines a type `hf` that comes with the following functionality:

- The function `HF :: hf set ⇒ set` that converts a finite set of HF sets into an HF set.
- The usual set operations such as equality (=), membership (∈), union (⊔), intersection (⊓), and difference (−) are defined.
- Finally, the empty set coincides with the ordinal 0, so it is denoted by `0 :: hf`.

Equipped with the above, we define the interpretation functions

- $\mathtt{I}_{\mathtt{st}}$ `:: ('a ⇒ hf) ⇒ 'a pset_term ⇒ hf` and
- $\mathtt{I}_{\mathtt{sa}}$ `:: ('a ⇒ hf) ⇒ 'a pset_atom ⇒ hf`

in the standard way, i.e. by mapping each syntactic construct to the corresponding operation on HF sets and interpreting variables with respect to a given valuation function `M :: 'a ⇒ hf`. For the concrete definition we refer to the formalisation.

We write `M` $\models \phi$ for the judgement that the formula ϕ holds under the valuation function `M`. The implementation of $\models$ coincides with the interpretation function of Nipkow [20]. As usual, we call a formula ϕ *satisfiable* if there exists a model `M` with `M` $\models \phi$. Otherwise, we say that ϕ is *unsatisfiable*.

3 A Tableau Calculus for MLSS

We formalise the tableau calculus for **MLSS** as described by Cantone and Zarba [9]. Inspired by the formalisation of a tableau calculus for hybrid logic by From [16], we use lists to represent the branches of the tableau tree. Note that we add formulas to the front of the list during branch expansion, so `last b` for a branch `b` is always the formula we are trying to disprove with the tableau. We sometimes call this formula the *initial formula*.

```
type_synonym 'a branch = 'a pset_fm list
```

We lift the functions `vars` and `subterms` to branches in the expected way.

In the standard tableau calculus for propositional logic as Fitting [15] describes it, a branch is called *closed* if it contains both the negation of a formula and the formula itself; conversely, it is called *open* if it is not closed. For **MLSS**, we extend the notion of closedness with three additional rules; the first two are straightforward while the last one states that a branch is closed when the branch contains a membership cycle $\mathtt{t}_0 \in_{\mathtt{s}} \mathtt{t}_1$, $\mathtt{t}_1 \in_{\mathtt{s}} \mathtt{t}_2$, …, $\mathtt{t}_{\mathtt{k}} \in_{\mathtt{s}} \mathtt{t}_0$.

Table 1. Linear expansion rules. All rules except the double negation rule coincide with the original paper [9]. For brevity, we omit the rules for $\sqcap_s$ and $-_s$.

Propositional Rules			Rules for $\sqcup_s$		
$p \wedge q$	$\Longrightarrow$	$p,\ q$	$s \notin_s t_1 \sqcup_s t_2$	$\Longrightarrow$	$s \notin_s t_1,\ s \notin_s t_2$
$\neg(p \vee q)$	$\Longrightarrow$	$\neg p,\ \neg q$	$s \in_s t_1$	$\Longrightarrow$	$s \in_s t_1 \sqcup_s t_2$
$p \vee q,\ \neg p$	$\Longrightarrow$	q	$s \in_s t_2$	$\Longrightarrow$	$s \in_s t_1 \sqcup_s t_2$
$p \vee q,\ \neg q$	$\Longrightarrow$	p	$s \in_s t_1 \sqcup_s t_2,$ $s \notin_s t_1$	$\Longrightarrow$	$s \in_s t_2$
$\neg(p \wedge q),\ p$	$\Longrightarrow$	$\neg q$	$s \in_s t_1 \sqcup_s t_2,$ $s \notin_s t_2$	$\Longrightarrow$	$s \in_s t_1$
$\neg(p \wedge q),\ q$	$\Longrightarrow$	$\neg p$	$s \notin_s t_1,\ s \notin_s t_2$	$\Longrightarrow$	$s \notin_s t_1 \sqcup_s t_2$
$\neg(\neg p)$	$\Longrightarrow$	p			
Rules for $\sqcap_s$			Rules for $-_s$		
⋮			⋮		
Rules for Single			Rules for $=_s$		
	$\Longrightarrow$	$s \in_s \mathtt{Single}\ s$	$t_1 =_s t_2,\ l$	$\Longrightarrow$	$l\{t_2/t_1\}$
$s \in_s \mathtt{Single}\ t$	$\Longrightarrow$	$s =_s t$	$t_1 =_s t_2,\ l$	$\Longrightarrow$	$l\{t_1/t_2\}$
$s \notin_s \mathtt{Single}\ t$	$\Longrightarrow$	$s \neq_s t$	$s_1 \in_s t,\ s_2 \notin_s t$	$\Longrightarrow$	$s_1 \neq_s s_2$

```
inductive bclosed :: 'a branch ⇒ bool where
  ⟦ ϕ ∈ set b; ¬ϕ ∈ set b ⟧ ⟹ bclosed b
| (t ∈s ∅) ∈ set b ⟹ bclosed b
| (t ≠s t) ∈ set b ⟹ bclosed b
| ⟦ member_cycle cs; set cs ⊆ set b ⟧ ⟹ bclosed b
```

```
abbreviation bopen b ≡ ¬ bclosed b
```

A tableau is called *closed* if all of its branches are closed.

3.1 Linear Expansion Rules

The calculus considers two kinds of branch expansion rules: *linear* and *branching* rules. As the name suggests, branching rules lead to the creation of new branches in the tableau while linear rules only extend a branch b with new formulas $b' = [\psi_1,\ldots,\psi_n]$, which we denote by $b' \rhd b$. Table 1 shows the linear expansion rules. Note that in the first two rules for $=_s$, l is a literal occurring in the branch. Furthermore, the term-for-term substitution l{s/t} is restricted to the top-level set terms of l, i.e. the set terms that occur directly under one of the atom constructors $\in_s$ or $=_s$; for example, given the literal

$$l = \neg((s \sqcup_s u) -_s s =_s s \sqcup_s u)$$

we have

Table 2. Branching expansion rules. We write ϕ for `last b` here. All rules coincide with the original paper [9] so we only show an illustrative subset.

Rule	Precondition	Subsumption condition
$\dfrac{}{\texttt{p} \mid \neg\,\texttt{p}}$	$\texttt{p} \vee \texttt{q} \in \texttt{set b}$	$\texttt{p} \in \texttt{set b} \vee$ $\neg\ \texttt{p} \in \texttt{set b}$
$\dfrac{}{\texttt{s} \in_{\texttt{s}} \texttt{t}_1 \mid \texttt{s} \notin_{\texttt{s}} \texttt{t}_1}$	$(\texttt{s} \in_{\texttt{s}} \texttt{t}_1 \sqcup_{\texttt{s}} \texttt{t}_2) \in \texttt{set b}$ $\texttt{t}_1 \sqcup_{\texttt{s}} \texttt{t}_2 \in \texttt{subterms}\ \phi$	$(\texttt{s} \in_{\texttt{s}} \texttt{t}_1) \in \texttt{set b}$ $\vee\ (\texttt{s} \notin_{\texttt{s}} \texttt{t}_1) \in \texttt{set b}$
$\vdots$	$\vdots$	$\vdots$
$\dfrac{}{\begin{array}{l} \texttt{Var x} \in_{\texttt{s}} \texttt{t}_1 \\ \texttt{Var x} \notin_{\texttt{s}} \texttt{t}_2 \end{array} \middle\| \begin{array}{l} \texttt{Var x} \notin_{\texttt{s}} \texttt{t}_1 \\ \texttt{Var x} \in_{\texttt{s}} \texttt{t}_2 \end{array}}$	$(\texttt{t}_1 \neq_{\texttt{s}} \texttt{t}_2) \in \texttt{set b}$ $\texttt{t}_1 \in \texttt{subterms}\ \phi$ $\texttt{t}_2 \in \texttt{subterms}\ \phi$ $\texttt{x} \notin \texttt{vars b}$	$\exists \texttt{s}.\ (\texttt{s} \in_{\texttt{s}} \texttt{t}_1) \in \texttt{set b}$ $\wedge\ (\texttt{s} \notin_{\texttt{s}} \texttt{t}_2) \in \texttt{set b}$ $\vee$ $\exists \texttt{s}.\ (\texttt{s} \notin_{\texttt{s}} \texttt{t}_1) \in \texttt{set b}$ $\wedge\ (\texttt{s} \in_{\texttt{s}} \texttt{t}_2) \in \texttt{set b}$

$$\begin{aligned} &(\neg\,((\texttt{s} \sqcup_{\texttt{s}} \texttt{u})\ -_{\texttt{s}}\ \texttt{s}\ =_{\texttt{s}}\ \texttt{s} \sqcup_{\texttt{s}} \texttt{u}))\{\texttt{t/s} \sqcup_{\texttt{s}} \texttt{u}\} \\ = {}&\neg\,((\texttt{s} \sqcup_{\texttt{s}} \texttt{u})\ -_{\texttt{s}}\ \texttt{s}\ =_{\texttt{s}}\ \texttt{t}). \end{aligned}$$

A more crucial restriction of the linear rules is that no new subterm may be created by their application; for instance, the second rule for $\sqcup_{\texttt{s}}$ is

$$\texttt{s} \in_{\texttt{s}} \texttt{t}_1 \Longrightarrow \texttt{s} \in_{\texttt{s}} \texttt{t}_1 \sqcup_{\texttt{s}} \texttt{t}_2,$$

which formally represents

$$(\texttt{s} \in_{\texttt{s}} \texttt{t}_1) \in \texttt{set b} \Longrightarrow [\texttt{s} \in_{\texttt{s}} \texttt{t}_1 \sqcup_{\texttt{s}} \texttt{t}_2] \rhd \texttt{b},$$

and may only be used under the condition $\texttt{t}_1 \sqcup_{\texttt{s}} \texttt{t}_2 \in \texttt{subterms (last b)}$. The purpose of this restriction is to prevent unbounded expansion of the branch. In fact, we give an explicit upper bound for the number of formulas in a branch in Sect. 7.

Due to boundedness, repeated expansion with linear rules eventually results in a *linearly saturated* branch, i.e. a branch where no application of linear rules would produce new formulas.

```
definition lin_sat b ≡ ∀b'. b' ⊳ b ⟶ set b' ⊆ set b
```

Finally, we remark that the original paper [9] is missing the last propositional rule dealing with double negation. This rule is required for completeness, though, considering that the branch $[\neg\neg\neg\texttt{p},\ \texttt{p},\ \neg\neg\neg\texttt{p} \wedge \texttt{p}]$ is saturated—neither linear nor branching rules apply—and open, but there clearly is no model for the initial formula $\neg\neg\neg\texttt{p} \wedge \texttt{p}$.

3.2 Branching Rules

After running out of linear rules to apply, only the branching rules shown in Table 2 remain. A rule is applicable if its *precondition* is met and, to prevent unnecessary branching, if it is not subsumed as indicated by the *subsumption condition*. These rules create multiple branches in the tableau, so we represent the different possibilities `bs'` to expand a branch `b` as a set and write `bs' ⊳ b`. Accordingly, we get a new branch `b' @ b` in the tableau for each `b' ∈ bs'`.

A linearly saturated branch where no further branching is possible is called a *saturated* branch.

```
definition sat b ≡ lin_sat b ∧ (∄bs'. bs' ⊳ b)
```

Note that even branching rules are defined such that they never create new subterms, except for the last rule that adds a new variable to the branch. These variables serve to manifest an inequality; hence, we call them *witnesses*.

```
definition wits b ≡ vars b - vars (last b)
```

4 A Decision Procedure for MLSS

The mechanics of the decision procedure are typical for a procedure based on a tableau calculus: it decides the satisfiability of a given formula ϕ by determining whether the formula has a closed tableau. More specifically, it initialises the tableau with the singleton branch `[`ϕ`]` and checks whether this branch can be expanded to a closed tableau.

We only discuss the abstract specification here and refer the reader to the formalisation for the executable specification. The implementation uses a couple of features of Isabelle/HOL's function package: instead of defining the function via pattern matching, we specify the equations of the function as conditional rewrite rules. This requires us to prove that the assumptions of the equations are non-overlapping, which is done by automation. The other concern is that Isabelle/HOL requires functions to be total, so a recursive function needs to terminate for it to be well-defined; nevertheless, the termination proof is separated from the definition of the function for modularity. The function package maintains the soundness of the definition by introducing a so-called domain predicate `mlss_proc_branch_dom` which characterises the arguments for which the function terminates. Each equation of the function is guarded by an assumption that the predicate holds for the argument. In Sect. 7, we will show that the domain predicate holds for the context in which the function `mlss_proc_branch` is called in. Before we go into more detail on how the termination is proved, we discuss the definition of the function, as shown below.

```
function mlss_proc_branch :: 'a branch ⇒ bool where
  ¬ lin_sat b ⟹ mlss_proc_branch b =
  mlss_proc_branch ((SOME b'. b' ⊳ b ∧
                                  set b ⊂ set (b' @ b)) @ b)
```

```
| ⟦ lin_sat b; bclosed b ⟧ ⟹ mlss_proc_branch b = True
| ⟦ ¬ sat b; bopen b; lin_sat b ⟧ ⟹ mlss_proc_branch b =
  (∀b' ∈ (SOME bs. bs ⊳ b). mlss_proc_branch (b' @ b))
| ⟦ lin_sat b; sat b ⟧ ⟹ mlss_proc_branch b = bclosed b
```

```
definition mlss_proc :: 'a pset_fm ⇒ bool where
  mlss_proc φ ≡ mlss_proc_branch [φ]
```

The purpose of the function is to determine whether we can expand a given branch to a closed tableau. As stated before, we first use linear expansion rules in order to prevent premature branching; to this end, we recursively expand the branch with linear rules until the branch is linearly saturated. Note that we use Hilbert's ε-operator in the form of SOME[1] to choose some rule that actually adds new formulas to the branch. As soon as the branch is linearly saturated, we terminate if the branch is closed as the second equation shows. Otherwise, we choose an applicable branching rule and recursively check whether all newly created branches can be closed. The final equation applies once no further branch expansion is possible, in which case we just test for closedness of the branch.

The procedure mlss_proc then calls mlss_proc_branch with a singleton branch [ϕ] to determine the satisfiability of a given formula ϕ.

Thus, we use mlss_proc_branch is only on branches that result from applying the expansion rules. We call this kind of branch *well-formed*. In the definition below, the expression b' ⊳* b denotes that b' is one of the branches that results from applying (potentially zero) expansion rules to b.

```
definition wf_branch b ≡ ∃φ. b ⊳* [φ]
```

We use this notion in Sect. 7 to state an upper bound for the cardinality of well-formed branches. The upper bound justifies the termination of the decision procedure. Before we come to that, though, we prove soundness and completeness in Sect. 6 and 5, respectively. In Sect. 7, we also show that both properties easily transfer to mlss_proc, which, together with termination, establishes that it is a decision procedure.

5 Completeness of the Calculus

For completeness of the calculus, we need to show that every unsatisfiable formula has a closed tableau or, conversely, that the formula is satisfiable if there is a saturated and open branch in the tableau. To facilitate inductive reasoning, we show a stronger statement by constructing a model M such that M $\models \phi$ for all $\phi \in$ set b. At the core of the model, there is a *realisation* function that maps set terms to sets of type hf. A subset of the witnesses, which we call *pure* witnesses, receives special treatment from the realisation function for reasons that will become apparent in Sect. 5.1. The collection of set terms of a branch can thus be partitioned into two collections, as defined below.

[1] In the formalisation, the function mlss_proc_branch is actually parametrised by choice functions to allow for refinement.

```
definition pwits :: 'a branch ⇒ 'a set where
  pwits b ≡ {c ∈ wits b. ∀t ∈ subterms (last b).
              AT (Var c =s t) ∉ set b ∧ AT (t =s Var c) ∉ set b}
```

```
definition subterms' :: 'a branch ⇒ 'a pset_term set where
  subterms' b ≡ subterms (last b) ∪ Var ‘ (wits b - pwits b)
```

We aim to construct a syntactic model that we derive from the membership literals $s \in_s t$ in the branch. To this end, we construct a graph whose vertices are the disjoint union of the sets above and there is an edge from `s` to `t` in the graph if, and only if, $s \in_s t$ is in `b`. Note that we use Noschinski's graph library [22] which represents a graph as a record of vertices, arcs (directed edges), and two functions `tail` and `head` that map an arc to its source and target vertex, respectively.

```
definition bgraph b ≡ let vs = Var ‘ pwits b ∪ subterms' b
  in ⦇ verts = vs, arcs = {(s, t). (s ∈s t) ∈ set b},
       tail = fst, head = snd ⦈
```

The realisation function is defined relative to this graph. As mentioned before, the realisation function treats the pure witnesses differently than the rest of the set terms. The function evaluates terms in the latter set in accordance to the structure of the graph, i.e. the realisation of a vertex is defined as the union of the realisations of the parent vertices. For the former set, we choose a function `I` that assigns the pure witnesses pairwise distinct sets with cardinality greater than that of the vertices. We can always choose such a function since we assume an infinite universe of variables. Then, we return the singleton set `HF {I x}`, which, together with the cardinality constraint, guarantees that realisations are distinct between pure witnesses themselves as well as between pure witnesses and set terms. The notation $u \rightarrow_G s$ in the definition below indicates that there is an edge from `u` to `s` in the graph `G`.

```
abbreviation parents G s ≡ {u. u →G s}
```

```
function realise :: 'a pset_term ⇒ V where
  x ∈ Var ‘ pwits b ⟹ realise x = HF {I x}
| x ∈ subterms' b
  ⟹ realise t = HF {realise ‘ parents (bgraph b) s}
| x ∉ verts G ⟹ realise x = 0
```

Again, we need to ensure that the assumptions of the equations are non-overlapping and that the function terminates. The former is taken care of by automation, leaving us to prove termination. The assumption that `b` is open implies that there are no membership cycles, thus `bgraph b` is acyclic. Furthermore, the graph is finite by definition. Thus, we can use the cardinality of the set of ancestors as a measure that decreases in each recursive call.

Before we prove that the realisation function constitutes a model in Sect. 5.2, we will first explain the significance of the pure witnesses.

5.1 Characterisation of the Pure Witnesses

Recall that the pure witnesses of a branch `b` are those witnesses that are not related to other subterms in `last b` by equality. In the context of a well-formed branch, we can strengthen this characterisation to any set term and, in addition, we also get that there is no membership literal where a pure witness is on the right-hand side. Intuitively speaking, the realisation of a pure witness does not depend on the realisation of any other set term.

```
lemma lemma_2:
  assumes wf_branch b and c ∈ pwits b
  shows (Var c =_s t) ∉ set b and (t =_s Var c) ∉ set b
    and (t ∈_s Var c) ∉ set b
```

So why are pure witnesses treated differently? According to the definition of `realise`, it would evaluate the pure witnesses would to the empty set `0 :: hf`, were they not treated separately. To see that this is a problem, consider the branch `b = [Var s` $\neq_s$ `Var t, Var t` $\neq_s$ `Var u]` which expands to several open and saturated branches, one of which is

```
[Var x ≠_s Var y, Var x ∈_s Var s, Var x ∉_s Var t,
                  Var y ∈_s Var t, Var y ∉_s Var u] @ b
```

for some fresh `x` and `y`. Assigning both `Var x` and `Var y` a value of `0` would contradict the literal `Var x` $\neq_s$ `Var y`. To prevent this, we assign the pure witnesses pairwise different values.

The proof of `lemma_2` is more technical than interesting so we refer the reader to the formalisation.

5.2 Realisation of an Open Branch

Remember that for completeness, we need to show that the realisation function for an open and saturated branch `b` actually constitutes a model for all formulas in the branch. We start by verifying that the realisation function models all literals in the branch; more formally, the following propositions hold:

(1) We have `realise s` $\in$ `realise t` if it holds that `s` $\in_s$ `t` is in `b`.
(2) We have `realise s` $=$ `realise t` if `s` $=_s$ `t` is in `b`.
(3) We have `realise s` $\neq$ `realise t` if `s` $\neq_s$ `t` is in `b`.
(4) We have `realise s` $\notin$ `realise t` if it holds that `s` $\notin_s$ `t` is in `b`.

To illustrate the usefulness of `lemma_2`, we prove Proposition (2). The proofs of all propositions translate well into Isabelle, so we refer to the original paper [9] for the remaining proofs.

Proof. (Proof of Proposition (2)*).* Assume that `s` $=_s$ `t` is in `b`. If there exists a `c` $\in$ `pwits b` where `s = Var c` or `t = Var c`, we arrive at a contradiction due to `lemma_2`. Therefore, both `s` $\in$ `subterms' b` and `t` $\in$ `subterms' b` must hold. Now, assume for contradiction that `realise s` $\neq$ `realise t`. Without loss of generality—the other case is symmetric—we obtain an $\mathtt{e}$ such that $\mathtt{e} \in \mathtt{realise\ s}$ and $\mathtt{e} \notin \mathtt{realise\ t}$. Considering that $\mathtt{s} \in \mathtt{subterms'\ b}$ and the definition of $\mathtt{realise}$, we obtain a $\mathtt{d}$ with $\mathtt{e = realise\ d}$ and $\mathtt{d} \longrightarrow_{\mathtt{bgraph\ b}} \mathtt{s}$. This, in turn, yields that $\mathtt{d} \in_{\mathtt{s}} \mathtt{s}$ must be in $\mathtt{b}$. Together with the assumption $(\mathtt{s} =_{\mathtt{s}} \mathtt{t}) \in \mathtt{set\ b}$ and the saturation of $\mathtt{b}$, it follows that $\mathtt{d} \in_{\mathtt{s}} \mathtt{t}$ must also be in $\mathtt{b}$. But then we have $\mathtt{realise\ d} \in \mathtt{realise\ t} \longleftrightarrow \mathtt{e} \in \mathtt{realise\ t}$ using Proposition (1), which is a contradiction to the assumption $\mathtt{e} \notin \mathtt{realise\ t}$.

We now lower the results on literals to set terms. All of the proofs are straightforward so we refer the reader to the formalisation.

(a) It holds that $\mathtt{realise}\ \emptyset = \mathtt{0}$.
(b) Let $\star_{\mathtt{s}} \in \{\sqcup_{\mathtt{s}}, -_{\mathtt{s}}, \sqcap_{\mathtt{s}}\}$. If the term $\mathtt{s} \star_{\mathtt{s}} \mathtt{t}$ occurs in $\mathtt{subterms\ b}$, then

$$\mathtt{realise\ (s} \star_{\mathtt{s}} \mathtt{t) = realise\ s} \star \mathtt{realise\ t}.$$

(c) If $\mathtt{Single\ t} \in \mathtt{subterms\ b}$, then

$$\mathtt{realise\ (Single\ t) = HF\ \{realise\ t\}}.$$

The final step for obtaining a proper model is to connect the realisation function to the semantics as defined in Sect. 2. For set terms, we can use the Propositions (a)–(c) to prove the lemma below by induction on $\mathtt{t}$.

```
lemma assumes t ∈ subterms b
      shows Iₛₜ (λx. realise (Var x)) t = realise t
```

Lifting the above result to formulas yields the coherence of $\mathtt{b}$, as the original paper [9] calls it. The proof is a tedious but straightforward induction on the the size of the formulas.

```
lemma coherence:
  assumes φ ∈ set b shows (λx. realise (Var x)) ⊨ φ
```

The coherence property finishes the proof of completeness of the calculus as it gives us a model for every formula in an open and saturated branch.

6 Soundness of the Calculus

A tableau calculus is sound if the corresponding formula is unsatisfiable for any closed tableau. We prove the following two properties to establish soundness:

(1) It is impossible to satisfy all formulas in a closed branch simultaneously.
(2) The expansion rules maintain satisfiability.

We formalise the first property in Isabelle below.

```
lemma bclosed_sound:
  assumes bclosed b shows ∃φ ∈ set b. M ⊭ φ
```

Proof. It is clear that, for any `s`, neither does `M` model $\texttt{s} \in \emptyset$ nor $\texttt{s} \neq_\texttt{s} \texttt{s}$. Furthermore, no model can satisfy both ϕ and $\neg\phi$ at the same time. Lastly, a membership cycle is impossible since the membership relation of `hf` is well-founded.

We are left with showing that both linear and branching expansion rules preserve satisfiability. As for the linear rules, a straightforward proof by case analysis on `b' ▷ b` suffices to obtain the lemma below.

```
lemma lexpands_sound:
  assumes b' ▷ b and ϕ ∈ set b' and ⋀ψ. ψ ∈ set b ⟹ M ⊨ ψ
  shows M ⊨ ϕ
```

A similar argument would work for the branching rules if it were not for the last rule adding new variables. Those variables need to be assigned specific values; hence, we modify the model as shown in the proof below.

```
lemma bexpands_sound:
  assumes bs' ⊳ b and ⋀ψ. ψ ∈ set b ⟹ M ⊨ ψ
  shows ∃M'. ∃b' ∈ bs'. ∀ψ ∈ set (b' @ b). M' ⊨ ψ
```

Proof. We only consider the case where `bs' ⊳ b` was proved by applying the last branching expansion rule to $\texttt{s} \neq_\texttt{s} \texttt{t}$ for some `s` and `t`. We have

```
  bs' = {[Var x ∈s s, Var x ∉s t], [Var x ∈s t, Var x ∉s s]}
```

for some fresh variable `x`. Since $\texttt{s} \neq_\texttt{s} \texttt{t}$ is in `b`, we have that $\texttt{I}_{\texttt{st}}\ \texttt{M s} \neq \texttt{I}_{\texttt{st}}\ \texttt{M t}$ because `M` is a model. Without loss of generality, this inequality manifests itself through some `y` with $\texttt{y} \in \texttt{I}_{\texttt{st}}\ \texttt{M s}$ and $\texttt{y} \notin \texttt{I}_{\texttt{st}}\ \texttt{M t}$. We update `M` to map `x` to `y` to obtain the assignment `M'`. Note that `M'` is still a model for formulas in `b` because `x` is fresh with respect to `b`. Furthermore, it is also a model for the first branch in `bs'`, which finishes the proof.

7 Total Correctness of the Decision Procedure

We first demonstrate the termination of the procedure for well-formed branches, i.e. every well-formed branch is in the domain of `mlss_proc_branch`. To this end, we derive an upper bound for the number of distinct formulas in a branch whose proof we omit here for brevity. We should point out that this bound is not to be construed as the complexity of the procedure as it may create exponentially many branches in general.

```
lemma card_wf_branch_ub:
  assumes wf_branch b
  shows |set b| ≤ 2 * |subfms (last b)| + 16 * |subterms (last b)|^4
```

Remember that `mlss_proc_branch` only applies a linear expansion rule to a branch if the application results in new formulas. Moreover, the subsumption conditions of the branching expansion rules ensure that each of the newly created branches contain new formulas. Ultimately, we conclude that the procedure must terminate for well-formed branches because the number of formulas increases in each step but is also bounded.

```
lemma assumes wf_branch b shows mlss_proc_branch_dom b
```

The above lemma allows us to utilise the computation induction rule of `mlss_proc_branch` on well-formed branches, which we use to prove soundness and completeness. As both proofs are essentially an application of soundness, respectively completeness, of the calculus, we refer the reader to the formalisation.

```
lemma mlss_proc_branch_complete:
  fixes b :: ’a branch
  assumes wf_branch b and ¬ mlss_proc_branch b
  assumes infinite (UNIV :: ’a set)
  shows ∃M. M ⊨ last b
```

```
lemma mlss_proc_branch_sound:
  assumes wf_branch b and ∀ψ ∈ set b. M ⊨ ψ
  shows ¬ mlss_proc_branch b
```

To finish the proof of total correctness, note that every singleton branch is trivially well-formed; thus, termination, completeness, and soundness easily transfer to `mlss_proc`.

```
theorem mlss_proc_complete:
  fixes ϕ :: ’a pset_fm
  assumes ¬ mlss_proc ϕ and infinite (UNIV :: ’a set)
  shows ∃M. M ⊨ ϕ
```

```
theorem mlss_proc_sound:
  assumes M ⊨ ϕ shows ¬ mlss_proc ϕ
```

8 Dealing with Urelements

In the introduction, we stated the goal of integrating `mlss_proc` as a tactic into Isabelle. For this to work, we must map every branch expansion rule to a corresponding theorem in Isabelle/HOL. This is straightforward for all expansion rules except for the last branching expansion rule. To illustrate, suppose that we are to disprove a statement of the form

```
s ≠ (t :: ’a) ∧ s ∈ (A :: ’a set) ∪ B ∧ ...
```

in Isabelle/HOL. By way of reification, we convert this to a formula of the shape

$$\texttt{s'} \neq_{s} \texttt{t'} \wedge \texttt{s'} \in_{s} \texttt{A'} \sqcup_{s} \texttt{B'} \wedge \ldots$$

in our set syntax for some `s’`, `t’`, `A’`, and `B’`. When we apply the decision procedure to this formula, it might return a tableau proof that contains an application of the last branching rule to (`s’` $\neq_s$ `t’`) ∈ `set b`. This results in two branches, one of which is [`Var x` $\in_s$ `s’`, `Var x` $\notin_s$ `t’`] @ `b`; however, there is no matching rule in Isabelle/HOL since `s` and `t` are not sets.

To deal with this problem, we formalise a lightweight type system as displayed in Fig. 1. The type of a set term in this system is just a natural number which we call level. Intuitively speaking, the level `l` means that the corresponding term `t` in Isabelle/HOL has type

$$\texttt{'a}\ \underbrace{\texttt{set} \ldots \texttt{set}}_{\texttt{l}\text{ times}}$$

for some `'a`. Note that the constructor $\emptyset$ now receives an additional argument indicating the level of each instance of $\emptyset$.

Moreover, the typing judgement extends to set atoms by matching up the levels of its component set terms.

Ultimately, we define $\Gamma \vdash \phi \equiv \forall \texttt{a} \in \texttt{atoms}\ \phi.\ \Gamma \vdash \texttt{a}$ in order to type formulas.

We can now define the urelements with respect to a formula. An urelement is a set term whose corresponding type in Isabelle/HOL might not be a set.

```
definition urelem :: 'a pset_fm ⇒ 'a pset_term ⇒ bool where
  urelem φ t ≡ ∃Γ. Γ ⊢ φ ∧ Γ ⊢ t : 0
```

Using this definition, we make two changes to the specification of the calculus: (1) First and foremost, we require that neither `s` nor `t` is an urelement in the precondition of the last branching expansion rule. (2) As mentioned above, we add an argument to the $\emptyset$ constructor. This argument is only used for the typing judgement; it has no impact on the semantics.

Soundness, of course, is not affected by these changes but we have to make a few amendments to maintain completeness: (1) The first equation of `realise` now also must account for the urelements. In particular, it has to ensure that urelements receive pairwise different values unless they are related through equality atoms. This does not affect pure witnesses since they can not be related through equality atoms due to `lemma_2`. (2) We must adjust the completeness proof in those places where it directly refers to the definition of `realise` to account for the case where a given term is an urelement. (3) The completeness theorem receives the additional assumption that $\Gamma \vdash \phi$ holds for the initial formula ϕ.

$$\frac{}{\Gamma \vdash \emptyset\ \texttt{n} : \texttt{Suc n}} \qquad \frac{}{\Gamma \vdash \texttt{Var x} : \Gamma\ \texttt{x}} \qquad \frac{\Gamma \vdash \texttt{t} : \texttt{l}}{\Gamma \vdash \texttt{Single t} : \texttt{Suc l}}$$

$$\frac{\star_{\texttt{s}} \in \{\sqcup_{\texttt{s}}, \sqcap_{\texttt{s}}, -_{\texttt{s}}\} \quad \Gamma \vdash \texttt{s} : \texttt{l} \quad \Gamma \vdash \texttt{t} : \texttt{l} \quad \texttt{l} \neq 0}{\Gamma \vdash \texttt{s} \star_{\texttt{s}} \texttt{t} : \texttt{l}}$$

$$\frac{\Gamma \vdash \texttt{s} : \texttt{l} \quad \Gamma \vdash \texttt{t} : \texttt{l}}{\Gamma \vdash \texttt{s} =_{\texttt{s}} \texttt{t}} \qquad \frac{\Gamma \vdash \texttt{s} : \texttt{l} \quad \Gamma \vdash \texttt{t} : \texttt{Suc l}}{\Gamma \vdash \texttt{s} \in_{\texttt{s}} \texttt{t}}$$

Fig. 1. The type system for set terms and atoms.

(4) For the completeness proof, we must show that the typing judgement is invariant under branch expansion.

The modifications above ensure that the proof can be replayed through Isabelle/HOL. To actually use the calculus, we must determine the urelements of the initial formula ϕ, though. In other words, we have to implement an inference algorithm for our lightweight type system. The algorithm is, in essence, a simplified version of Hindley-Milner type inference so it has the same two phases: it generates constraints using syntax directed rules and then passes them to a constraint solver.

Since we are only interested in the level of a term, we can encode all constraints into the theory of 0, the successor function `S`, and equality (but no disequality). Note that constraints of the form $\texttt{l} \neq \texttt{0}$ can be replaced by $\texttt{l} = \texttt{S i}$ with `i` being a fresh variable. A solver for this theory is straightforward to implement and verify; nevertheless, we have to be careful that it computes the minimum assignment Γ from variables to levels that fulfils the constraints. This guarantees that a set term `t` is not an urelement if, and only if, $\Gamma\ \texttt{t} > \texttt{0}$. Conversely, all terms `s` with $\Gamma\ \texttt{s} = \texttt{0}$ are urelements.

9 Conclusion and Future Work

We developed a formalisation of a tableau calculus for a quantifier-free fragment of set theory called **MLSS** based on a paper by Cantone and Zarba [9]. The formalisation includes an abstract description of a decision procedure that builds on the calculus. To make the decision procedure compatible with Isabelle/HOL, we extended the calculus with a lightweight type system while maintaining completeness. We also refined the abstract specification to an executable specification from which code can be generated.

In future work, we plan to implement an efficient executable specification in the style of a worklist algorithm. This specification should also generate certificates that can be replayed through Isabelle's inference kernel to facilitate the integration of the procedure into Isabelle.

Acknowledgements. The author thanks Kevin Kappelmann and Tobias Nipkow for their comments on a draft version of this paper and the anonymous referees for their thorough reviews.

References

1. Beckert, B., Hartmer, U.: A tableau calculus for quantifier-free set theoretic formulae. In: de Swart, H. (ed.) TABLEAUX 1998. LNCS (LNAI), vol. 1397, pp. 93–107. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-69778-0_16
2. Bentzen, B.: A Henkin-style completeness proof for the modal logic S5. In: Baroni, P., Benzmüller, C., Wáng, Y.N. (eds.) CLAR 2021. LNCS (LNAI), vol. 13040, pp. 459–467. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-89391-0_25

3. Blanchette, J.C., Popescu, A., Traytel, D.: Soundness and completeness proofs by coinductive methods. J. Autom. Reason. **58**(1), 149–179 (2016). https://doi.org/10.1007/s10817-016-9391-3
4. Cantone, D.: A fast saturation strategy for set-theoretic tableaux. In: Galmiche, D. (ed.) TABLEAUX 1997. LNCS, vol. 1227, pp. 122–137. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0027409
5. Cantone, D., Longo, C., Asmundo, M.N.: A decision procedure for a two-sorted extension of multi-level syllogistic with the cartesian product and some map constructs. In: Faber, W., Leone, N. (eds.) Italian Conference on Computational Logic, CEUR Workshop Proceedings, vol. 598, CEUR-WS.org (2010). http://ceur-ws.org/Vol-598/paper11.pdf
6. Cantone, D., Omodeo, E.G., Policriti, A.: The automation of syllogistic. J. Autom. Reasoning **6**(2), 173–187 (1990). https://doi.org/10.1007/BF00245817. ISSN 0168-7433
7. Cantone, D., Omodeo, E.G., Policriti, A.: Set Theory for Computing - From Decision Procedures to Declarative Programming with Sets. Monographs in Computer Science. Springer, Heidelberg (2001). https://doi.org/10.1007/978-1-4757-3452-2
8. Cantone, D., Schwartz, J.T., Zarba, C.G.: A decision procedure for a sublanguage of set theory involving monotone, additive, and multiplicative functions. Electron. Notes Theor. Comput. Sci. **86**(1), 49–60 (2003). https://doi.org/10.1016/S1571-0661(04)80652-2. International Workshop on First-Order Theorem Proving
9. Cantone, D., Zarba, C.G.: A new fast tableau-based decision procedure for an unquantified fragment of set theory. In: Caferra, R., Salzer, G. (eds.) FTP 1998. LNCS (LNAI), vol. 1761, pp. 126–136. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46508-1_8
10. Cantone, D., Zarba, C.G.: A tableau-based decision procedure for a fragment of set theory involving a restricted form of quantification. In: Murray, N.V. (ed.) TABLEAUX 1999. LNCS (LNAI), vol. 1617, pp. 97–112. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48754-9_12
11. Chaieb, A.: Verifying mixed real-integer quantifier elimination. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 528–540. Springer, Heidelberg (2006). https://doi.org/10.1007/11814771_43
12. Chaieb, A., Nipkow, T.: Verifying and reflecting quantifier elimination for presburger arithmetic. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 367–380. Springer, Heidelberg (2005). https://doi.org/10.1007/11591191_26
13. Doczkal, C., Smolka, G.: Completeness and decidability results for CTL in Coq. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 226–241. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_15
14. Ferro, A., Omodeo, E.G., Schwartz, J.T.: Decision procedures for elementary sublanguages of set theory. I. Multi-level syllogistic and some extensions. Commun. Pure Appl. Math. **33**(5), 599–608 (1980). https://doi.org/10.1002/cpa.3160330503
15. Fitting, M.: Semantic Tableaux and Resolution. Springer, New York (1996). https://doi.org/10.1007/978-1-4612-2360-3_3
16. From, A.H.: Formalizing a Seligman-style tableau system for hybrid logic. Archive of Formal Proofs (2019). ISSN 2150-914x. https://isa-afp.org/entries/Hybrid_Logic.html. Formal proof development
17. From, A.H.: Synthetic completeness. Archive of Formal Proofs (2023). ISSN 2150-914x. https://isa-afp.org/entries/Synthetic_Completeness.html. Formal proof development

18. From, A.H., Blackburn, P., Villadsen, J.: Formalizing a seligman-style tableau system for hybrid logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12166, pp. 474–481. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_27
19. From, A.H., Schlichtkrull, A., Villadsen, J.: A sequent calculus for first-order logic formalized in Isabelle/HOL. In: Monica, S., Bergenti, F. (eds.) Proceedings of the 36th Italian Conference on Computational Logic, CEUR Workshop Proceedings, vol. 3002, pp. 107–121. CEUR-WS.org (2021). http://ceur-ws.org/Vol-3002/paper7.pdf
20. Nipkow, T.: Linear quantifier elimination. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 18–33. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_3
21. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL–A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
22. Noschinski, L.: Graph theory. Archive of Formal Proofs (2013). ISSN 2150-914x. https://isa-afp.org/entries/Graph_Theory.html. Formal proof development
23. Paulson, L.C.: The hereditarily finite sets. Archive of Formal Proofs (2013). ISSN 2150-914x. https://isa-afp.org/entries/HereditarilyFinite.html. Formal proof development
24. Stevens, L.: MLSS decision procedure. Archive of Formal Proofs (2023). ISSN 2150-914x. https://isa-afp.org/entries/MLSS_Decision_Proc.html. Formal proof development
25. Stevens, L., Nipkow, T.: A verified decision procedure for orders in Isabelle/HOL. In: Hou, Z., Ganesh, V. (eds.) ATVA 2021. LNCS, vol. 12971, pp. 127–143. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88885-5_9

An Experimental Pipeline for Automated Reasoning in Natural Language (Short Paper)

Tanel Tammet[1](✉), Priit Järv[1], Martin Verrev[1], and Dirk Draheim[2]

[1] Applied Artificial Intelligence Group, Tallinn University of Technology, Tallinn, Estonia
{tanel.tammet,priit.jarv1,martin.verrev}@taltech.ee

[2] Information Systems Group, Tallinn University of Technology, Tallinn, Estonia
dirk.draheim@taltech.ee

Abstract. We describe an experimental implementation of a logic-based end-to-end pipeline of performing inference and giving explained answers to questions posed in natural language. The main components of the pipeline are semantic parsing, integration with large knowledge bases, automated reasoning using extended first order logic, and finally the translation of proofs back to natural language. While able to answer relatively simple questions on its own, the implementation is targeting research into building hybrid neurosymbolic systems for gaining trustworthiness and explainability. The end goal is to combine machine learning and large language models with the components of the implementation and to use the automated reasoner as an interface between natural language and external tools like database systems and scientific calculations.

1 Introduction

Question answering and inference using natural language is a classic A.I. area, with a long history of little success using symbolic methods, able to solve only small problems with a limited structure. The recent machine learning (ML) systems, in particular, the Large Language Model (LLM) implementations of the BERT and GPT families are, in contrast, often able to give satisfactory answers to nontrivial questions.

However, the current LLMs are neither trustworthy nor explainable. They have a well-known tendency of "hallucinating", i.e. giving wrong answers and inventing actually nonexistent entities and facts. The problems of explicitly controlling the output and giving explanations for the solutions appear to be very hard for LLMs. An optimistic view of LLMs suggests that end-to-end learning can be improved to overcome these issues, while a more pessimistic view suggests that the problems are inherent and stem from the lack of an internal world model. The proponents of the latter view propose to build hybrid neurosymbolic systems, combining machine learning and symbolic methods of various kinds.

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 509–521, 2023.
https://doi.org/10.1007/978-3-031-38499-8_29

Indeed, the research in the field of neurosymbolic systems has become quite active. The recent survey [14] points to a wider interest in connecting natural language systems to external software like databases and scientific calculations.

Using logic for natural language inference (NLI) in combination with ML may potentially alleviate the problems with LLMs and provide a glue to connect external systems to natural language interfaces. However, using logic directly for processing natural language is hard, for a number of reasons:

- Semantic parsing, i.e. translating natural language to logic, is extremely hard due to the highly complex and exception-rich nature of natural language.
- Existing knowledge bases of "common sense" do not cover a critical mass of the basic understanding of the world even a small child possesses.
- Classical first order reasoning itself cannot cope with contradictory knowledge items, probabilistic or uncertain information and exceptions to rules.
- Finding logic-based proofs often requires long proofs and the huge knowledge base causes a quick combinatorial explosion of the search space.

The motivation behind the research described in the paper is the following hypothesis: all the main problems described above can be alleviated by using ML techniques tailored separately for each particular problem. The current paper does not introduce any ML techniques for the problems above. The goal of our system is to serve as a backbone for research into combining the symbolic methods with ML. Our hypothesis is that by gradual improvement and combination of the existing symbolic subsystems with ML techniques it is possible to eventually build a question answering system which has enough power, trustworthiness and explainability to be practically useful in various application areas.

In other words, the envisioned end goal of this research is neither to replace LLMs nor to verify their output, but to develop systems combining LLMs and symbolic reasoning for specific areas where it is feasible to build sets of domain-specific rules and factual databases.

2 Related Work

Here we will only consider projects building a full NLP inference system. The performance of older pure symbolic or logic-based methods like LogAnswer [7] remained at the level of specific toy examples and never achieved capabilities required for wider applicability. The long-running CYC project [22], although having several successes, did not succeed with its original stated goals, which is often used as an argument against symbolic systems.

A popular area for language processing is converting human queries to SQL or SPARQL queries. These systems typically do not handle rules expressed in natural language. The projects closest to ours use reasoners with a relatively limited capacity, like BRAID [12], which uses extended SLD+ reasoner with probabilistic rules and fuzzy unification, CASPR [18], which uses an ASP reasoner incorporating default logic, NatPro [1,2], which uses a Natural Logic prover. The latter is the only such project we know to be publicly available: https://github.com/kovvalsky/prove_SICK_NL.

The majority of research in neurosymbolic reasoning for natural language combines ML with weak forms of symbolic systems, typically taxonomies and triple graph knowledge bases like ConceptNet [25]. We approach the problem from the less common direction: starting from the symbolic/reasoning side and moving towards ML. There are already a few research projects combining ML with reasoning in quantified first order logic, although we are not aware of any such systems being publicly available. Noteworthy projects involving quantified logic are SQuARE [4], BRAID [12] and STAR [21]. The recent work in using large language models (LLM) mapping informal proofs to formal Isabelle [17] proof sketches guiding an automated prover [34] and using LLMs directly to generate Isabelle code [11] shows clear promise in combining LLMs with provers.

3 Natural Language Inference and Question Answering

The described pipeline is able to handle both the natural language inference (NLI) tasks (given a premise, determine whether a given hypothesis is true, false or indeterminate) and the closely related question answering tasks of finding a specific object matching a given criterion.

We will use a few simple examples throughout the paper. The expected answer to the first example *"If an animal likes honey, then it is probably a bear. Most bears are big, although young bears are not big. John is an animal who likes honey. Mike is a young bear. Who is big?"* is *"Likely John"*. The expected answer to the second example *"The length of the red car is 4 m. The length of the black car is 5 m. The length of the red car is less than 5 m?"* is *"True"*.

It is worth noting that these examples are solved correctly by the current (May 2023) versions of GPT: ChatGPT using the text-davinci-002 model and the API using the gpt-3.5-turbo and gpt-4 models: moreover, they are able to give a satisfactory explanation of the reasoning behind the answers. However, if we insert additional irrelevant information to the first example, our system still finds the expected answer, while none of the GPT models above give a correct answer: *"If an animal likes honey, then it is probably a bear. Most bears are big, although young bears are not big. John is an animal who likes honey. Mike is a young bear. Mike can eat a lot. Penguins are birds who cannot fly. John took the block from the colored table. The table was really nice. The robot arm lifted a blue block from the table. Who is big?"*.

Similarly, when we modify the second example by using meaningless words and adding irrelevant text, our system finds the expected answer, while all the referred GPT models give confusing answers: *"The length of the barner is 200000000 m. The length of the red foozer is 312435 m. Most barners are 1000000 m long. Sun is larger than the moon. John saw the sun rising over an enormous foozer. A huge robot filled the sky. The length of the red foozer is less than 312546 m?"* However, the answers given by GPT versions may vary over time, i.e. experiments with GPT are not reproducible.

4 The Question Answering Pipeline

Our system is publicly available at http://github.com/tammet/nlpsolver. It requires Linux and should be easy to install. The implementation consists of four main software systems. The pipeline driver calls the external Stanza parser [20] from Stanford, giving a Universal Dependencies (UD, see [5]) graph, then runs the semantic parser on the UD graph, calls the reasoner, and finally builds a natural language answer along with the explanation built from the proofs given by the reasoner. The pipeline driver, parser and answer construction components consist of over 400 Kbytes of Python code. Before running the solver, a small Python server component has to be started, to initialize the external UD parser Stanza and read a commonsense knowledge base into shared memory. For reasoning the pipeline calls our commonsense reasoner GK, written in C: this is the largest and the most complex part of the pipeline. There is a separate Python program for regression tests, along with several Python files containing sub-tests, currently over 1600 separate NLI tasks. The pipeline driver is called from a command line, with a natural language text and question as a command line argument, plus a number of optional arguments to control the behaviors like the amount of output.

4.1 Semantic Parsing

The parser takes English strings of natural language text as input and outputs extended clausified first-order logic formulas encoded in JSON as proposed in JSON-LD-LOGIC [29]. The main extension is adding numerical confidence to clauses and implementing default logic [23] by including special literals to encode exceptions, as presented in our papers [28] and [27].

Parsing consists of a number of phases, each adding new structural details to the results of the previous phases. For the most part, the phases are implemented procedurally, without using explicit transformation rules: we found that the more complex aspects of translation cannot be easily expressed with the help of simple transformation rules. In particular, the correct interpretation of a sentence depends heavily on previous sentences and a collected database of objects which have been talked about.

Conversion to Universal Dependencies (UD) Format. We use the external Stanza parser to get the UD format dependency graphs from input sentences. Stanza itself uses pretrained neural models. We first preprocess English strings to avoid several typical mistakes of the Stanza conversion, and then use Stanza to get the UD graph. The graph is then fed to our small set of simplifying transformations returning a simplified text, which is again fed to Stanza to get the final UD graph. The simplification phase reduces the amount of complexities and edge case handling necessary in the UD-to-logic converter, and is a prime candidate for experimenting with using LLMs for simplifications.

Converting UD to Logic. One of the strengths of UD representation given by Stanza is a high level of detail. The first subphase of conversion is restructuring the UD graph to a semi-logical representation explicating the outward logical structure around the subject/verb, object/verb or subject/verb/object tuples. The following subphases attach different kinds of properties to words. For example, the outmost structure constructed for the sentence "Most bears are big, although young bears are not big." is
`[and, svo[bear,be,big], svo[bear,be,big]]` which is then extended to
`[and, svo[bear,be,big], svo[[props,young,bear],be,big]]`.
The words in these structures are key-value objects containing both the initial UD information and additional details added during the phases.

The next subphase results in the extended logic in a non-clausified form, i.e. using explicit quantifiers. The conversion uses the previous structure recursively, taking into account the details of the original UD structure to find additional critical information like articles, negation, different kinds of quantifiers etc. We follow the approach of Davidsonian semantics, introducing event identification variables, while not taking the neo-Davidsonian path of splitting all relations to their minimal components (see [33])

For the coreference resolution we calculate the weighted heuristic scores for all candidate words, using also taxonomies of Wordnet. Another inherently complex task is determining whether a noun stands for a concrete object or should be quantified over. Importantly, any object detected is stored in a special data structure with new information about the object possibly added as the parsing process proceeds.

Let us consider an example sentence "John is a nice animal who likes honey." It would be first converted to a conjunction of three formulas

$$\mathtt{isa}(\mathtt{animal}, \mathtt{c1\_John})$$
$$\mathtt{prop}(\mathtt{nice}, \mathtt{c1\_John}, \mathtt{generic}, \mathtt{generic}, \mathtt{ctxt}(\mathtt{Pres}, 1))$$
$$\mathtt{def0}(\mathtt{c1\_John})$$
$$\forall \mathtt{S}\ (\mathtt{def0}(\mathtt{c1\_John}) \leftrightarrow$$
$$\exists \mathtt{X}\ \mathtt{isa}(\mathtt{honey}, \mathtt{X})\ \&\ (\exists \mathtt{A}\ \mathtt{do2}(\mathtt{like}, \mathtt{c1\_John}, \mathtt{X}, \mathtt{A}, \mathtt{ctxt}(\mathtt{Pres}, \mathtt{S}))))$$

The system determined that in this sentence "John" refers to a concrete object and immediately created a Skolem constant `c1_John`, storing it for possible later use and extension. Here it also created a new definition `def0` for encoding the complex property of "John": liking honey. The properties of objects like given in the second formula above also encode the intensity of the property (slightly/very) and the comparative class: for example, saying "John is a very large animal ..." would create $\mathtt{prop}(\mathtt{large}, \mathtt{c1\_John}, 3, \mathtt{animal}, \mathtt{ctxt}(\mathtt{Pres}, 1))$. The constant `generic` indicates that intensity is not known or that the property is not comparative, i.e. does not relate to a specific class. The term $\mathtt{ctxt}(\mathtt{Pres}, 1)$ encodes contextual aspects: the present tense and a concrete situation number in a possible sequence of situations created by different actions. The variable `A` in the last formula is an identifier of an action, which can be given additional properties, like place, time or assistive objects of an action, in the Davidsonian style.

In the representations above we have omitted the information about confidence and the possibility of exceptions. Indeed, the sentence we looked at is considered to be certain and without exceptions. However, the first part of the sentence "Most bears are big, although young bears are not big" attaches confidence 0.85 to the formula and includes a *blocker literal* encoding an exception in the sense of default logic, along with the comparative priority of the blocker:

$$\begin{aligned}&\mathtt{0.85}: \forall \mathtt{X}\, \mathtt{isa}(\mathtt{bear},\mathtt{X}) \rightarrow \\ &\quad (\mathtt{prop}(\mathtt{big},\mathtt{X},\mathtt{generic},\mathtt{generic},\mathtt{ctxt}(\mathtt{Pres},\mathtt{1})))\vee \\ &\quad \mathtt{block}(\mathtt{h}(\mathtt{bear},\mathtt{1}),\mathtt{neg}(\mathtt{prop}(\mathtt{big},\mathtt{X},\mathtt{generic},\mathtt{generic},\mathtt{ctxt}(\mathtt{Pres},\mathtt{1}))))\end{aligned}$$

The blocker literals are used by the GK prover to recursively check the proof candidates found, with dimishing time limits: GK uses a part of a given time limit to attempt to prove each blocker literal in the proof. Whenever a blocker is proved, the candidate proof containing the blocker is considered invalid and thus discarded; see [27] for details.

The system is also able to handle simpler questions involving sizes of sets, like "An animal had two strong legs. The animal had a strong leg?", "John has three big nice cars. John has two big cars?", and measures, like "The length of the red car is 4 m. The length of the black car is 5 m. The length of the red car is less than 5 m?". We use terms encoding the sets and measures: for example, the first sentence of the last question is translated to a formula containing a standard equality predicate, an integer and several properties involving the measure term, including the main statement $4 = \mathtt{count}(\mathtt{measure1}(\mathtt{length},\mathtt{c1\_car},\mathtt{meter},\mathtt{ctxt}(\mathtt{Pres},\mathtt{1}))$

Instance Generation. In order to answer questions without indicating concrete objects, like "Adult bears are large animals. Cats are small animals. Who is a large animal?" we need constants representing an anonymous instance of a class, essentially a "default adult bear", a "default bear" and a "default cat". For each such object the system generates a constant along with the formulas indicating its class and properties, enabling the system to produce an answer "An adult bear".

Question Handling. Actual questions like "Who is big?" or "The length of the red car is less than 5 m?" require special handling. The automated reasoner GK used in the pipeline employs the well-known *answer predicate* technique to construct and output the required substitution term. All the variables in the question formula will be instantiated and output, potentially resulting in a large combination of different answers. The "Who is big?" question will be first translated to $\exists \mathtt{X},\mathtt{Y},\mathtt{Z}\, \mathtt{prop}(\mathtt{big},\mathtt{X},\mathtt{generic},\mathtt{Y},\mathtt{Z})$ indicating that we are not restricting the "bigness" or context in the question. However, we do not want to enumerate different "bigness" values or contexts in the answer, thus we wrap the formula into a definition (say, `def2`) over a single variable `X`, and search for different substitutions into `def2(X)` only. Asking questions about location and time is implemented by constructing a number of questions over relations "near", "on", "at", etc.

Clausification and Simplification. The system contains a clausifier skolemizing the formulas and converting these to a conjunctive normal form. The clausification phase also performs several simplifications, some of which are possible due to the known properties of the constructed formulas. Since nontrivial formulas may be converted into several clauses, the clausifier decides how to spread the numeric confidence of the formula and the exception literals in the formula into the clauses.

4.2 Integration with Knowledge Bases

The knowledge base provides the world model of our reasoning system. To answer the query "Tweety is a bird. Can Tweety fly?", the system needs to have the background knowledge that birds can fly. We construct the knowledge base (KB) using default logic rules augmented with numeric confidences. A small part of the knowledge base forms a core world model and is built by hand, while the bulk of the knowledge is integrated automatically from existing common sense knowledge (CSK) sources as described in [10].

We have integrated eight published knowledge graphs: ConceptNet [25], WebChild [30], Aristo TupleKB [15], Quasimodo [24], Ascent++ [16], UnCommonSense [3], ATOMIC$^{20}_{20}$ [9] and ATOMIC^{10x} [32]. These CSK sources are collections of relation triples. The majority of the sources contain natural language clauses or fragments in the triple elements. We have built a specialized pattern matching semantic parser to convert the relations to first order logic rules with the default logic extensions and estimated numeric confidence. The full knowledge base contains 18.5 million rules, with over 15 million of those are related to taxonomy: inferring a property or an event from the class of an entity.

4.3 Automated Reasoning

We use our automated reasoner GK to solve the problems generated by semantic parser. The reasoner uses both the parser output and a selected subset of the world knowledge to solve the questions. Wordnet taxonomies are used to solve the precedence problem of exceptions. Large datasets are parsed, indexed and kept in shared memory for quick re-use. GK is built on top of a conventional high-performance resolution-based reasoner GKC [26] for conventional first order logic. Thus GK inherits most of the capabilities and algorithms of GKC. The main additional features of GK are following:

- Using a well-known answer clause mechanism for finding a number of different answers, with a configurable limit.
- Finding expected proofs even if a knowledge base is inconsistent. Basically, GK only accepts proofs which contain a clause originating from the question.
- Searching for both a proof of the question and a negation of the question/negation of each concrete answer.
- Estimating the numeric confidence in the statements derived from knowledge bases containing uncertain contrary and supporting evidence obtained from different sources.
- Handling exceptions by implementing default logic via recursively deepening iterations of searches with diminishing time limits.

- Performing reasoning by analogy via employing known similarity scores of words along with exceptions.

The first four features are covered in our previous paper [28] and the following two are covered in [27]. The word similarity handling is currently in an experimental phase: the initial experiments show that a naive implementation creates an unmanageable search space explosion, and thus a layered approach is necessary.

As a simple example of the basic features, consider sentences “John is nice. John is not nice. Mike is nice. Steve is not nice.” GK output to the parsed versions of the following questions will directly lead to these answers: “John is nice?”: “Unknown”, “Mike is nice?”: “True”, “Mike is not nice?”: “False”, “Who is nice?”: “Mike”, “Who is not nice?”: “Steve”. For a slightly more complex example, consider the earlier “If an animal likes honey, then it is probably a bear. Most bears are big, although young bears are not big. John is an animal who likes honey. Mike is a young bear. Who is big?”. GK will output the following proof in JSON, where we have removed quotation marks and a number of steps:

```
{result:answer found,

answers:[
{
answer:[[$ans,some_bear]],
blockers:
  [[$block,[$,bear,1],[$not,[prop,big,some_bear,$generic,$generic,[$ctxt,Pres,1]]]]],
confidence:0.85,
positive proof:
[
...,
[7,[mp,[5,1],6,fromgoal,0.85],
  [[$block,[$,bear,1],[$not,[prop,big,some_bear,$generic,$generic,[$ctxt,Pres,1]]]],
  [$ans,some_bear]]]
]},
{
answer:[[$ans,c1_John]],
blockers:[[$block,[$,bear,1],[$not,[prop,big,c1_John,$generic,$generic,[$ctxt,Pres,1]]]],
          [$block,[$,animal,3],[$not,[isa,bear,c1_John]]]],
confidence:0.765,
positive proof:
[
[1,[in,frm_10,axiom,0.85],
   [[$block,[$,bear,1],[$not,[prop,big,?:X,$generic,$generic,[$ctxt,Pres,1]]]],
    [prop,big,?:X,$generic,$generic,[$ctxt,Pres,1]],
    [-isa,bear,?:X]]],
[2,[in,frm_9,axiom,0.9],
   [[$block,[$,animal,3],[$not,[isa,bear,?:X]]],
    [-do2,like,?:X,?:Y,?:Z,[$ctxt,Pres,1]],
    [-isa,honey,?:Y],[-isa,animal,?:X],[isa,bear,?:X]]],
...,
[18,[mp,[1,2],[17,1],fromaxiom,0.765],
  [[$block,[$,bear,1],[$not,[prop,big,c1_John,$generic,$generic,[$ctxt,Pres,1]]]],
  [$block,[$,animal,3],[$not,[isa,bear,c1_John]]],
  [prop,big,c1_John,$generic,$generic,[$ctxt,Pres,1]]]],
...,
[21,[in,frm_30,goal,1],[[-$def2,?:X],[$ans,?:X]]],
[22,[mp,[20,2],21,fromgoal,0.765],
  [[$block,[$,bear,1],[$not,[prop,big,c1_John,$generic,$generic,[$ctxt,Pres,1]]]],
   [$block,[$,animal,3],[$not,[isa,bear,c1_John]]],
   [$ans,c1_John]]]
]}
]}
```

Observe that we get two answers. The following NLP pipeline step removes the generic `[[$ans,some_bear]]`, since the more informative `[[$ans,c1_John]]` is available. Here both proofs contain only positive parts, although in the general case we may find both a positive and a negative proof, each with their own confidences. GK will throw away both the clauses produced during search and the final answers which have a summary confidence below a configurable threshold. GK will also throw away proofs which do not contain a goal clause. The confidences stemming from input sentences like "Most bears are big ..." are taken from our ad-hoc mapping of words like "most" to numeric values. By default, "normal" rule sentences are given a confidence below one and include a blocker literal for allowing exceptions.

The answers contain blocker literals, which have been recursively checked by separate proof searches before the final proof is accepted by GK. The details of these failed searches are not shown in the final proof. Had we included the sentence "John is not big" in our example, then the proof of the first blocker of the main answer would have been found, thus disqualifying the proof and leaving us with the final answer "Likely a bear.".

4.4 Answers and Explanations in Natural Language

Answers and explanations are generated from the proof, with additional details taken from the database of objects along with their properties as detected during semantic parsing. While some of the principles were described in the previous section, there are two major tasks to perform: give a suitably detailed representation of objects in a proof (say, select between "a car", "a red car", "the red car", "Mike's car" etc.) and create a grammatically correct and easy-to-understand textual representation of clauses. The system translates clauses in a proof one-to-one to English sentences, as exemplified by the explanation generated from the previously presented proof:

```
Likely john:
Confidence 76%.
Sentences used:
(1) If an animal likes honey, then it is probably a bear.
(2) Most bears are big, although young bears are not big.
(3) John is an animal who likes honey.
(4) Who is big?
Statements inferred:
(1) If X is a bear, then X is big. Confidence 85%. Why: sentence 2.
(2) If X does like Y and Y is a honey and X is an animal, then X is a bear.
    Confidence 90%. Why: sentence 1.
(4) If John has a property def1, then John does like cs4. Why: sentence 3.
...
(18) John is big. Confidence 76%. Why: statements 1, 17.
...
(21) If X matches the query, then X is an answer. Why: the question.
(22) John is an answer. Confidence 76%. Why: statements 20, 21.
```

5 Performance and the Test Set

The system has miserable performance on most well-known natural language inference or question answering benchmarks, the majority of which are oriented towards machine learning. As an exception, the performance on the anti-machine-learning question set HANS [13] is ca 95%, in contrast to the ca 60% performance of LLM systems before the GPT3 family (random choice would give 50% performance). The loss of 5% of HANS is due to the wrong UD parses chosen by Stanza.

However, the system is able to solve almost all of the demonstration examples of the Allen AI ProofWriter system https://proofwriter.apps.allenai.org/ and is able to solve inference problems the current LLM systems cannot, like the examples presented in the introduction. For regression testing we have built a set of ca 1600 simple questions with answers, structured over different types of capabilities. This test set may be of use for people working towards similar goals.

The runtime for the small examples presented in the paper is ca 0.5 s on a Linux laptop with a graphics card usable by Stanza. Of this time, Stanza UD parsing takes ca 0.17 s, UD to logic takes ca 0.04 s, and the rest is spent by the reasoner. For more complex examples the reasoner may spend unlimited time, i.e. the question is rather how complex questions can be solved in a preconfigured time window. In case the size of the input problem is relatively small and a tiny world model suffices for the solution, the correct answer is found in ca 1–2 s. However, in case the system is given a large knowledge base (KB) with a size of roughly one gigabyte, and the answer actually depends on the KB, then the search space may explode and the system may fail to find answer in a reasonable time. Efficiently handling a very large knowledge base clearly requires suitable heuristics based on the semantics and interdependence of rules/facts in the KB.

6 Towards a Hybrid Neurosymbolic System

Although the scope of the sentences successfully parsed and questions answered could be improved by adding more and more specialized cases to the current system, the cost/benefit ratio of this work would rapidly decrease. We'll describe the most promising avenues of extending the system with ML hybridization as we currently see them.

Semantic Parsing. The two main approaches would be (a) end-to-end learning from sentences directly to extended logic as exemplified in [31], and (b) using existing LLMs or training specialized LLMs to perform simplification of sentences to the level where a hand-made semantic parser is able to convert the sentence to logic. Our initial experiments with the GPT models have shown that using a suitable prompt causes the LLMs to successfully split and simplify complex sentences.

Automated Reasoning. Despite being optimized for large knowledge bases and performing well in reasoning competitions on such problems, our system often fails to find nontrivial proofs in reasonable time in case a large knowledge base is used. The main approaches here would be (a) learning to find a proof, based on the experience of previous proofs (see [19] for an example), (b) using machine learning along with measures of semantic relatedness of formulas to the assumption and the question (see [6]) for an example), (c) using LLMs to predict intermediate results or relevant facts and rules. A significant boost in the terms of usability could be achieved by integrating external systems like databases and scientific computing with the automated reasoners.

The Knowledge Base. Publicly available knowledge bases do not focus on formalizing a basic world model, arguably critical for common-sense reasoning. It is possible that a core part needs to be built by hand. On the other hand, the existing knowledge bases along with large text corpuses can be extended by creating crucial new uncertain rules using both simpler statistical methods and more complex ML techniques: see [8] for a review.

7 Summary and Future Work

We have described an implementation of a full natural language inference and question answering pipeline built around an extended first order reasoner. The system is capable of understanding relatively simple sentences and giving reasonable answers to questions, including the types currently out of scope of the capabilities of LLMs. We plan to enhance the capabilities of the system by incorporating machine learning techniques to the components of pipeline, while keeping the overall architecture, including the semantic parser, word knowledge and a reasoner. At the time of this writing we are experimenting with using off-the-shelf LLMs without finetuning, but with a suitable prompt, to split and simplify complex sentences to a degree where our semantic parser is able to properly convert the meaning of the resulting sentences to logic.

References

1. Abzianidze, L.: Solving textual entailment with the theorem prover for natural language. Appl. Math. Inf. **25**(2), 1–15 (2020). https://www.viam.science.tsu.ge/Ami/2020_2/8_Lasha.pdf
2. Abzianidze, L., Kogkalidis, K.: A logic-based framework for natural language inference in Dutch. CoRR abs/2110.03323 (2021). https://arxiv.org/abs/2110.03323
3. Arnaout, H., Razniewski, S., Weikum, G., Pan, J.Z.: Uncommonsense: informative negative knowledge about everyday concepts. In: Hasan, M.A., Xiong, L. (eds.) Proceedings of the 31st ACM International Conference on Information & Knowledge Management, Atlanta, GA, USA, 17–21 October 2022, pp. 37–46. ACM (2022). https://doi.org/10.1145/3511808.3557484
4. Basu, K., Varanasi, S.C., Shakerin, F., Gupta, G.: Square: Semantics-based question answering and reasoning engine. CoRR abs/2009.09158 (2020). https://arxiv.org/abs/2009.10239
5. De Marneffe, M.C., Manning, C.D., Nivre, J., Zeman, D.: Universal dependencies. Comput. Linguist. **47**(2), 255–308 (2021)
6. Furbach, U., Krämer, T., Schon, C.: Names are not just sound and smoke: word embeddings for axiom selection. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 250–268. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_15
7. Furbach, U., Glöckner, I., Pelzer, B.: An application of automated reasoning in natural language question answering. AI Commun. **23**(2–3), 241–265 (2010)

8. Han, X., et al.: More data, more relations, more context and more openness: a review and outlook for relation extraction. In: Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing, pp. 745–758 (2020)
9. Hwang, J.D., et al.: (comet-) atomic 2020: on symbolic and neural commonsense knowledge graphs. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, pp. 6384–6392 (2021)
10. Järv, P., Tammet, T., Verrev, M., Draheim., D.: Knowledge integration for commonsense reasoning with default logic. In: Proceedings of the 14th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management - KEOD, pp. 148–155. INSTICC, SciTePress (2022). https://doi.org/10.5220/0011532200003335
11. Jiang, A.Q., et al.: Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. CoRR abs/2210.12283 (2022). https://arxiv.org/abs/2210.12283
12. Kalyanpur, A., Breloff, T., Ferrucci, D.A., Lally, A., Jantos, J.: Braid: Weaving symbolic and statistical knowledge into coherent logical explanations. CoRR abs/2011.13354 (2020). https://arxiv.org/abs/2011.13354
13. McCoy, T., Pavlick, E., Linzen, T.: Right for the wrong reasons: diagnosing syntactic heuristics in natural language inference. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, pp. 3428–3448. Association for Computational Linguistics (2019)
14. Mialon, G., et al.: Augmented language models: a survey. CoRR abs/2302.07842 (2023). https://arxiv.org/abs/2302.07842
15. Mishra, B.D., Tandon, N., Clark, P.: Domain-targeted, high precision knowledge extraction. Trans. Assoc. Comput. Linguist. **5**, 233–246 (2017). https://doi.org/10.1162/tacl_a_00058
16. Nguyen, T.P., Razniewski, S., Romero, J., Weikum, G.: Refined commonsense knowledge from large-scale web contents. IEEE Trans. Knowl. Data Eng. (2022). https://doi.org/10.1109/TKDE.2022.3206505
17. Paulson, L.C.: Isabelle: A Generic Theorem Prover. Springer, Cham (1994)
18. Pendharkar, D., Basu, K., Shakerin, F., Gupta, G.: An asp-based approach to answering natural language questions for texts. Theory Pract. Logic Programm. **22**(3), 419–443 (2022). https://arxiv.org/abs/2009.10239
19. Piepenbrock, J., Heskes, T., Janota, M., Urban, J.: Guiding an automated theorem prover with neural rewriting. In: Blanchette, J., Kovacs, L., Pattinson, D. (eds.) IJCAR 2022. Lecture Notes in Computer Science, vol. 13385, pp. 597–617. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_35
20. Qi, P., Zhang, Y., Zhang, Y., Bolton, J., Manning, C.D.: Stanza: A python natural language processing toolkit for many human languages. CoRR abs/2003.07082 (2020). https://arxiv.org/abs/2003.07082
21. Rajasekharan, A., Zeng, Y., Padalkar, P., Gupta, G.: Reliable natural language understanding with large language models and answer set programming. CoRR abs/2302.03780 (2023). https://arxiv.org/abs/2302.03780
22. Ramachandran, D., Reagan, P., Goolsbey, K.: First-orderized researchcyc: expressivity and efficiency in a common-sense ontology. In: AAAI Workshop on Contexts and Ontologies: Theory, Practice and Applications, pp. 33–40 (2005)
23. Reiter, R.: A logic for default reasoning. Artif. Intell. **13**(1–2), 81–132 (1980)
24. Romero, J., Razniewski, S., Pal, K., Pan, J.Z., Sakhadeo, A., Weikum, G.: Commonsense properties from query logs and question answering forums. In: Zhu, W.,

et al. (eds.) Proceedings of CIKM 2019 - the 28th ACM International Conference on Information and Knowledge Management, pp. 1411–1420. ACM (2019)
25. Speer, R., Chin, J., Havasi, C.: ConceptNet 5.5: an open multilingual graph of general knowledge. In: Singh, S.P., Markovitch, S. (eds.) Proc. of AAAI 2017 - the 31st AAAI Conference on Artificial Intelligence, pp. 4444–4451. AAAI (2017)
26. Tammet, T.: GKC: a reasoning system for large knowledge bases. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 538–549. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_32
27. Tammet, T., Draheim, D., Järv, P.: Gk: implementing full first order default logic for commonsense reasoning (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 300–309. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_18
28. Tammet, T., Draheim, D., Järv, P.: Confidences for commonsense reasoning. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 507–524. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_29
29. Tammet, T., Sutcliffe, G.: Combining JSON-LD with first order logic. In: 2021 IEEE 15th International Conference on Semantic Computing (ICSC), pp. 256–261. IEEE (2021)
30. Tandon, N., de Melo, G., Weikum, G.: Webchild 2.0 : fine-grained commonsense knowledge distillation. In: Bansal, M., Ji, H. (eds.) Proceedings of ACL 2017, System Demonstrations, pp. 115–120. Association for Computational Linguistics (2017). https://doi.org/10.18653/v1/P17-4020
31. Wang, C., Bos, J.: Comparing neural meaning-to-text approaches for Dutch. Comput. Linguist. Neth. **12**, 269–286 (2022)
32. West, P., et al.: Symbolic knowledge distillation: from general language models to commonsense models. CoRR abs/2110.07178 (2021). https://arxiv.org/abs/2110.07178
33. Winter, Y., Zwarts, J.: Event semantics and abstract categorial grammar. In: Kanazawa, M., Kornai, A., Kracht, M., Seki, H. (eds.) MOL 2011. LNCS (LNAI), vol. 6878, pp. 174–191. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23211-4_11
34. Wu, Y., et al.: Autoformalization with large language models. Adv. Neural. Inf. Process. Syst. **35**, 32353–32368 (2022)

Combining Combination Properties: An Analysis of Stable Infiniteness, Convexity, and Politeness

Guilherme V. Toledo[1](✉), Yoni Zohar[1], and Clark Barrett[2]

[1] Bar-Ilan University, Ramat Gan, Israel
guivtoledo@gmail.com, yoni.zohar@cs.tau.ac.il
[2] Stanford University, Stanford, USA
barrett@cs.stanford.edu

Abstract. We make two contributions to the study of theory combination in satisfiability modulo theories. The first is a table of examples for the combinations of the most common model-theoretic properties in theory combination, namely stable infiniteness, smoothness, convexity, finite witnessability, and strong finite witnessability (and therefore politeness and strong politeness as well). All of our examples are sharp, in the sense that we also offer proofs that no theories are available within simpler signatures. This table significantly progresses the current understanding of the various properties and their interactions. The most remarkable example in this table is of a theory over a single sort that is polite but not strongly polite (the existence of such a theory was only known until now for two-sorted signatures). The second contribution is a new combination theorem showing that in order to apply polite theory combination, it is sufficient for one theory to be stably infinite and strongly finitely witnessable, thus showing that smoothness is not a critical property in this combination method. This result has the potential to greatly simplify the process of showing which theories can be used in polite combination, as showing stable infiniteness is considerably simpler than showing smoothness.

Keywords: Satisfiability modulo theories · Theory combination · Theory politeness

1 Introduction

Theory combination focuses on the following problem: given procedures for determining the satisfiability of formulas over individual theories, can we find a procedure for the combined theory? One of the foundational results in this field is in Nelson and Oppen's paper [9], where the authors show how to combine theories with disjoint signatures as long as they are both stably infinite, i.e., for every quantifier-free formula that is satisfied in the theory, there is an infinite interpretation of the theory that satisfies it.

With the introduction of stable infiniteness was born the notion of identifying model-theoretic properties that enable theory combination. It soon became clear,

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 522–541, 2023.
https://doi.org/10.1007/978-3-031-38499-8_30

however, that this first step was insufficient, since some important theories with real-world applications (like the theories of bit-vectors and finite datatypes) turned out not to be stably infinite. Early attempts to find alternatives for stable infiniteness in theory combination included the introduction of gentle [5], shiny [12], and flexible [7] theories. We focus here on the notion of *politeness*, which forms the basis for theory combination in the state-of-the-art SMT solver cvc5 [1].

First considered in [10], polite theories were originally defined as those theories that are both smooth and finitely witnessable. Both notions are much harder to test for than stable infiniteness, but once a theory is known to be polite, it can be combined with any other theory, even non-stably-infinite ones.

A small problem in the proof of the main result of the paper was corrected in later work [6]. This paper introduces a slightly different, more strict, definition of politeness, together with a correct proof showing that polite theories can be combined with arbitrary theories. Following [4], we refer to theories satisfying the new definition as *strongly* polite, which is defined as being both smooth and *strongly finitely witnessable*; with that in mind, we call theories satisfying the earlier definition simply *polite*.

For some time, it was not known whether there exists a theory that is polite but not strongly polite. Then, in 2021 Sheng et al. [11] provided an example. This suggests the need for a more thorough analysis of properties such as stable infiniteness, smoothness, finite witnessability, and strong finite witnessability, as they appear to interact with each other in sometimes surprising or unforeseeable ways. We add to this list *convexity*, which was shown to be closely related to stable infiniteness in [2].

In this paper, we provide an exhaustive analysis, with examples whenever possible, of whether and how these properties can coexist. Some combinations are obviously impossible, such as a strongly finitely witnessable theory that is not finitely witnessable; the feasibility of other combinations is more elusive; for instance, it is initially unclear whether there can be a one-sorted, non-stably-infinite theory that is also not finitely witnessable (we show that this is also impossible). A main result is a comprehensive table describing what is known about all possible combinations of these properties.

During the course of filling the table, we were also able to improve polite combination: by making the involved proof slightly more difficult, we can simplify the main polite theory combination result: we show that in order to combine theories, it is enough for one theory to be stably infinite and strongly finitely witnessable; there is no need for smoothness. This result simplifies the process of qualifying a theory for polite combination, as showing stable infiniteness is considerably simpler than showing smoothness.

The paper is organized as follows. Section 2 defines the basic notions we will make use of throughout the paper. Section 3 proves several theorems showing the unfeasibility of certain combinations of properties. Section 4 describes the example theories that populate the feasible entries of the table. Section 5 offers a new combination theorem. And finally, Sect. 6 gives concluding remarks and directions for future work.[1]

[1] Due to space limitations, proofs are included in an appendix to [13].

$$\psi^{\sigma}_{\geq n} = \exists\, \overrightarrow{x}.\bigwedge_{1\leq i<j\leq n} \neg(x_i = x_j), \quad \psi^{\sigma}_{\leq n} = \exists\, \overrightarrow{x}.\forall\, y.\bigvee_{i=1}^{n} y = x_i, \quad \psi^{\sigma}_{=n} = \psi^{\sigma}_{\geq n} \wedge \psi^{\sigma}_{\leq n}$$

Fig. 1. Cardinality Formulas. $\overrightarrow{x}$ stands for $x_1, \ldots, x_n$.

2 Preliminary Notions

2.1 First-Order Signatures and Structures

A many-sorted signature Σ is a triple formed by a countable set $\mathcal{S}_{\Sigma}$ of *sorts*, a countable set of function symbols $\mathcal{F}_{\Sigma}$, and a countable set of predicate symbols $\mathcal{P}_{\Sigma}$ which contains, for every sort $\sigma \in \mathcal{S}_{\Sigma}$, an equality symbol $=_{\sigma}$ (often denoted by $=$); each function symbol has an arity $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$ and each predicate symbol an arity $\sigma_1 \times \cdots \times \sigma_n$, where $\sigma_1, \ldots, \sigma_n, \sigma \in \mathcal{S}_{\Sigma}$ and $n \in \mathbb{N}$. Each equality symbol $=_{\sigma}$ has arity $\sigma \times \sigma$. A signature with no function or predicate symbols other than equalities is called *empty*.

A many-sorted signature Σ is *one-sorted* if $\mathcal{S}_{\Sigma}$ has one element; we may refer to many-sorted signatures simply as signatures. Two signatures are said to be *disjoint* if they share only sorts and equality symbols.

We assume for each sort in $\mathcal{S}_{\Sigma}$ a distinct countably infinite set of variables, and define terms, literals, and formulas (atomic or not) in the usual way. If s is a function symbol of arity $\sigma \rightarrow \sigma$ and x is a variable of sort σ, we define recursively the term $s^k(x)$, for $k \in \mathbb{N}$, as follows: $s^0(x) = x$, and $s^{k+1}(x) = s(s^k(x))$. We denote the set of free variables of sort σ in a formula φ by $vars_{\sigma}(\varphi)$, and given $S \subseteq \mathcal{S}_{\Sigma}$, $vars_S(\varphi) = \bigcup_{\sigma\in S} vars_{\sigma}(\varphi)$ (we use $vars(\phi)$ as shorthand for $vars_{\mathcal{S}_{\Sigma}}$).

A *Σ-structure* $\mathcal{A}$ is composed of sets $\sigma^{\mathcal{A}}$ for each sort $\sigma \in \mathcal{S}_{\Sigma}$, called the *domain of* σ, equipped with interpretations $f^{\mathcal{A}}$ and $P^{\mathcal{A}}$ of the function and predicate symbols, in a way that respects their arities. Furthermore, $=^{\mathcal{A}}_{\sigma}$ must be the identity on $\sigma^{\mathcal{A}}$.

A *Σ-interpretation* $\mathcal{A}$ is an extension of a Σ-structure that also interprets variables, with the value of a variable x of sort σ being an element $x^{\mathcal{A}}$ of $\sigma^{\mathcal{A}}$; we will sometimes say that an interpretation $\mathcal{B}$ is an interpretation on a structure $\mathcal{A}$ (over the same signature) to mean that $\mathcal{B}$ has $\mathcal{A}$ as its underlying structure. We write $\alpha^{\mathcal{A}}$ for the interpretation of the term α under $\mathcal{A}$; if Γ is a set of terms, we define $\Gamma^{\mathcal{A}} = \{\alpha^{\mathcal{A}} : \alpha \in \Gamma\}$. We write $\mathcal{A} \vDash \varphi$ if $\mathcal{A}$ satisfies φ. A formula φ is called *satisfiable* if it is satisfied by some interpretation $\mathcal{A}$.

We shall make use of standard cardinality formulas, given in Fig. 1. $\psi^{\sigma}_{\geq n}$ is only satisfied by a structure $\mathcal{A}$ if $|\sigma^{\mathcal{A}}|$ is at least n, $\psi^{\sigma}_{\leq n}$ is only satisfied by $\mathcal{A}$ if $|\sigma^{\mathcal{A}}|$ is at most n, and $\psi^{\sigma}_{=n}$ is only satisfied by $\mathcal{A}$ if $|\sigma^{\mathcal{A}}|$ is exactly n. In one-sorted signatures, we may drop σ from the formulas, giving us $\psi_{\geq n}$, $\psi_{\leq n}$ and $\psi_{=n}$.

The following lemmas are generalizations of the standard compactness and downward Skolem-Löwenheim theorems of first-order logic to the many-sorted case. They are proved in [8].

Lemma 1 ([8]). *A set of formulas is satisfiable iff each of its finite subsets is satisfiable.*

Lemma 2 ([8]). *If a set of formulas is satisfiable, there exists an interpretation $\mathcal{A}$ which satisfies it and where $\sigma^{\mathcal{A}}$ is countable whenever it is infinite, for every sort σ.*

A *theory* $\mathcal{T}$ is a class of all Σ-structures that satisfy some set of closed formulas (formulas without free variables), called the *axiomatization* of $\mathcal{T}$ which we denote as $Ax(\mathcal{T})$; such structures will be called the *models* of $\mathcal{T}$, a model being called *trivial* when $\sigma^{\mathcal{A}}$ is a singleton for some sort σ in $\mathcal{S}_{\Sigma}$. A Σ-interpretation $\mathcal{A}$ whose underlying structure is in $\mathcal{T}$ is called a $\mathcal{T}$-interpretation. A formula is said to be $\mathcal{T}$*-satisfiable* if there is a $\mathcal{T}$-interpretation that satisfies it; a set of formulas is $\mathcal{T}$-satisfiable if there is a $\mathcal{T}$-interpretation that satisfies each of its elements. Two formulas are $\mathcal{T}$*-equivalent* when every $\mathcal{T}$-interpretation satisfies one if and only if it satisfies the other. We write $\vdash_{\mathcal{T}} \varphi$ and say that φ is $\mathcal{T}$-valid if $\mathcal{A} \vDash \varphi$ for every $\mathcal{T}$-interpretation $\mathcal{A}$. Let Σ_1 and Σ_2 be disjoint signatures; by $\Sigma = \Sigma_1 \cup \Sigma_2$, we mean the signature with the union of the sorts, function symbols, and predicate symbols of Σ_1 and Σ_2, all arities preserved. Given a Σ_1-theory $\mathcal{T}_1$ and a Σ_2-theory $\mathcal{T}_2$, the $\Sigma_1 \cup \Sigma_2$-theory $\mathcal{T} = \mathcal{T}_1 \oplus \mathcal{T}_2$ is the theory axiomatized by the union of the axiomatizations of $\mathcal{T}_1$ and $\mathcal{T}_2$.

2.2 Model-Theoretic Properties

Let Σ be a signature. A Σ-theory $\mathcal{T}$ is said to be *stably infinite* w.r.t. $S \subseteq \mathcal{S}_{\Sigma}$ if, for every $\mathcal{T}$-satisfiable quantifier-free formula ϕ, there exists a $\mathcal{T}$-interpretation $\mathcal{A}$ satisfying ϕ such that, for each $\sigma \in S$, $\sigma^{\mathcal{A}}$ is infinite. $\mathcal{T}$ is *smooth* w.r.t. $S \subseteq \mathcal{S}_{\Sigma}$ when, for every quantifier-free formula ϕ, $\mathcal{T}$-interpretation $\mathcal{A}$ satisfying ϕ, and function κ from S to the class of cardinals such that $\kappa(\sigma) \geq |\sigma^{\mathcal{A}}|$ for every $\sigma \in S$, there exists a $\mathcal{T}$-interpretation $\mathcal{B}$ satisfying ϕ with $|\sigma^{\mathcal{B}}| = \kappa(\sigma)$, for every $\sigma \in S$.

Theorem 1. *Let Σ be a signature, $S \subseteq \mathcal{S}_{\Sigma}$, and $\mathcal{T}$ a Σ-theory. If $\mathcal{T}$ is smooth w.r.t. S, then it is also stably infinite w.r.t. S.*

For a finite set of sorts S, finite sets of variables V_σ of sort σ for each $\sigma \in S$, and equivalence relations E_σ on V_σ, the arrangement on $V = \bigcup_{\sigma \in S} V_\sigma$ induced by $E = \bigcup_{\sigma \in S} E_\sigma$, denoted by δ_V or δ_V^E, is the quantifier-free formula given by $\delta_V = \bigwedge_{\sigma \in S} \left[\bigwedge_{xE_\sigma y}(x = y) \wedge \bigwedge_{x\overline{E_\sigma}y} \neg(x = y)\right]$, where $\overline{E_\sigma}$ denotes the complement of the equivalence relation E_σ.

A theory $\mathcal{T}$ is said to be *finitely witnessable* w.r.t. the set of sorts $S \subseteq \mathcal{S}_{\Sigma}$ when there exists a function *wit*, called a *witness*, from the quantifier-free formulas into themselves that is computable and satisfies for every quantifier-free formula ϕ: (i) ϕ and $\exists\, \overrightarrow{w}.\, wit(\phi)$ are $\mathcal{T}$-equivalent, where $\overrightarrow{w} = vars(wit(\phi)) \setminus vars(\phi)$; and (ii) if $wit(\phi)$ is $\mathcal{T}$-satisfiable, then there exists a $\mathcal{T}$-interpretation $\mathcal{A}$ satisfying $wit(\phi)$ such that $\sigma^{\mathcal{A}} = vars_\sigma(wit(\phi))^{\mathcal{A}}$ for each $\sigma \in S$. $\mathcal{T}$ is said to be *strongly finitely witnessable* if it has a strong witness *wit*, which has the properties of a witness with the exception of (ii), satisfying instead: (ii') given a finite set of variables V and an arrangement δ_V on V, if $wit(\phi) \wedge \delta_V$ is $\mathcal{T}$-satisfiable, then there exists a $\mathcal{T}$-interpretation $\mathcal{A}$ satisfying $wit(\phi) \wedge \delta_V$ such that $\sigma^{\mathcal{A}} = vars_\sigma(wit(\phi) \wedge \delta_V)^{\mathcal{A}}$ for all $\sigma \in S$.

From the definitions, the following theorem directly follows:

Theorem 2. *Let Σ be a signature, $S \subseteq \mathcal{S}_\Sigma$, and $\mathcal{T}$ a Σ-theory. If $\mathcal{T}$ is strongly finitely witnessable w.r.t. S then it is also finitely witnessable w.r.t. S.*

A theory that is both smooth and finitely witnessable w.r.t. (a set of sorts) $\mathcal{S}$ is said to be *polite* w.r.t. $\mathcal{S}$; a theory that is both smooth and strongly finitely witnessable w.r.t. $\mathcal{S}$ is called *strongly polite* w.r.t. $\mathcal{S}$. For theories over one-sorted empty signatures, we have the following theorem from [11]:

Theorem 3 **([11]).** *Every one-sorted theory over the empty signature that is polite w.r.t. its only sort is strongly polite w.r.t. that sort.*

A one-sorted theory $\mathcal{T}$ is said to be *convex* if, for any conjunction of literals ϕ and any finite set of variables $\{u_1, v_1, ..., u_n, v_n\}$, $\vdash_{\mathcal{T}} \phi \rightarrow \bigvee_{i=1}^{n} u_i = v_i$ implies $\vdash_{\mathcal{T}} \phi \rightarrow u_i = v_i$, for some $i \in [1, n]$.

Given a one-sorted theory $\mathcal{T}$, its *mincard* function takes a quantifier-free formula ϕ and returns the countable cardinal $\min\{|\sigma^{\mathcal{A}}| : \mathcal{A}$ is a $\mathcal{T}$-interpretation that satisfies $\phi\}$.[2]

Throughout this paper, we will use **SI** for stably infinite, **SM** for smooth, **FW** for finitely witnessable, **SW** for strongly finitely witnessable, and **CV** for convex.

3 Negative Results

If it were possible, we would present examples of every combination of properties using only the one-sorted empty signature, which is the simplest signature imaginable.

Of course, this is not always possible: smooth theories are necessarily stably infinite, and strongly finitely witnessable theories are obligatorily finitely witnessable. But there are several other connections we now proceed to show, which further restrict the combinations of properties that are possible.

In Sect. 3.1, we show that, under reasonable conditions, a convex theory must be stably infinite, while the reciprocal is also true over the empty signature. In Sect. 3.2, we show that over the empty one-sorted signature, theories that are not stably infinite are necessarily finitely witnessable (a somewhat counter-intuitive result, since we usually look for theories that are, simultaneously, smooth and strongly finitely witnessable) and, more importantly, that stably-infinite and strongly finitely witnessable one-sorted theories are also strongly polite.

[2] Note that this definition was generalized in two different ways to the many-sorted case in [4] and [10]. However, for our investigation, the single-sorted case is enough.

3.1 Stable-Infiniteness and Convexity

Convexity is typically defined over one-sorted signatures. Here we offer the following generalization to arbitrary signatures.

Definition 1. *A theory $\mathcal{T}$ is said to be convex w.r.t. a set of sorts $S \subseteq \mathcal{S}_\Sigma$ if, for any conjunction of literals ϕ and any finite set of variables $\{u_1, v_1, ..., u_n, v_n\}$ with sorts in S, if $\vdash_\mathcal{T} \phi \rightarrow \bigvee_{i=1}^{n} u_i = v_i$ then $\vdash_\mathcal{T} \phi \rightarrow u_i = v_i$, for some $i \in [1, n]$.*

If we assume, as it is often natural to, that our theories have no trivial models, then convexity implies stable infiniteness. This is true for the one-sorted case, as proved in [2], but also for the many-sorted case as we show here. The proof is similar, though here we need to account for several sorts at once. In particular, the proof relies on Lemma 1.

Theorem 4. *If a Σ-theory $\mathcal{T}$ is convex w.r.t. some set S of sorts and, for each $\sigma \in \mathcal{S}$, $\vdash_\mathcal{T} \psi^\sigma_{\geq 2}$, then $\mathcal{T}$ is stably infinite w.r.t. S.*

Reciprocally, we may also obtain convexity from stable infiniteness, but only over empty signatures.

Theorem 5. *Any theory over an empty signature that is stably infinite w.r.t. the set of all of its sorts is convex w.r.t. any set of sorts.*

As we shall see in Sect. 4, this result is tight: there are theories over non-empty signatures that are stably infinite but not convex.

3.2 More Connections

We next present more connections between the properties. First, over the one-sorted empty signature, a theory must be either stably infinite or finitely witnessable.

Theorem 6. *Every one-sorted, non-stably-infinite theory $\mathcal{T}$ with an empty signature is finitely witnessable w.r.t. its only sort.*

The following theorem shows that for one-sorted theories, strong politeness is a corollary of strong finite witnessability and stable infiniteness (rather than smoothness).

Theorem 7. *Every one-sorted theory that is stably infinite and strongly finitely witnessable w.r.t. its only sort is smooth, and therefore strongly polite w.r.t. that sort.*

Generalizing this theorem to the case of many-sorted signatures is left for future work.

Finally, by combining previous results, we can also get the following theorem, which relates stable infiniteness, strong finite witnessability, and convexity.

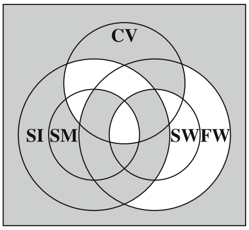


Fig. 2. A diagram of combinations over a one-sorted, empty signature: gray regions are empty.

Theorem 8. *A one-sorted theory $\mathcal{T}$ with an empty signature that is neither strongly finitely witnessable nor stably infinite w.r.t. its only sort cannot be convex.*

To summarize, while Theorem 4 is restricted to structures with no domains of cardinality 1, the remaining theorems of this section are not restricted to such structures. Theorem 5 applies to empty signatures, Theorem 7 applies to one-sorted signatures, and Theorems 6 and 8 apply to signatures that are both empty and one-sorted. Put together, we see that many combinations of properties for theories over a one-sorted empty signature are actually impossible. This is depicted in Fig. 2, in which all areas but the white ones are empty. For example, Theorem 6 shows that the area outside the **SI** and **FW** circles (representing theories that are neither stably infinite nor finitely witnessable) is empty, as every theory (over an empty one-sorted signature) must have one of these properties. Similarly, Theorem 8 further shows that within the **CV** (convex) circle, even more is empty, namely anything outside the **SI** and **SW** circles.

4 Positive Results

We now proceed to systematically address all possible combinations of stable-infiniteness, smoothness, finite witnessability, strong finite witnessability, and convexity.

The results are summarized in Table 1. Each row corresponds to a possible combination of properties, as determined by the truth values in the first five columns. For example, in the first row, the entries in the first five columns are all true, indicating that in this row, all theory examples must be stably-infinite, smooth, finitely witnessable, strongly finitely witnessable, and convex. The rest of the columns correspond to different possibilities for the theory signatures: either empty or non-empty, and either one-sorted or many-sorted. Again, looking at the first row, we see four different theories listed, one for each of the signature possibilities.

Some entries in the table list theorems instead of providing example theories. The listed theorems tell us that there do not exist any example theories for these entries. For example, lines 3 and 4 cannot provide examples over a one-sorted empty signature because of Theorem 3.

When an example is available, its name is given in corresponding cell of the table. The theories themselves are defined in Sect. 4.1 to 4.4. The examples on lines 25, 27 and 31 must have at least one structure with a trivial domain (i.e., a domain with exactly one element) because of Theorem 4.

Lines 9, 10, 13, and 14 cover theories that are stably infinite and strongly finitely witnessable but not smooth. We call these *unicorn theories* because we could not find any such theories, nor do we believe they exist, but (ignoring the obvious cases ruled out by Theorems 2, 5 and 7) we have no proof that they do not exist.

Definition 2. *A* unicorn theory *is stably infinite and strongly finitely witnessable but not smooth.*

Theorem 7 shows that there are no one-sorted unicorn theories. We believe it may be possible to provide a generalization of the upwards Löwenheim-Skolem theorem to many-sorted logic in such a way that it would prove the non-existence of unicorn theories, which leads to the following conjecture:

Conjecture 1. There are no unicorn theories.

Before defining the theories of Table 1, we introduce the following signatures.

Definition 3. *Σ_1 is the empty one-sorted signature with sort σ, Σ_2 is the empty two-sorted signature with sorts σ and σ_2, and Σ_s is the one-sorted signature with a single unary function symbol s.*

We now describe the theories: Sect. 4.1 describes the theories that are over the empty one-sorted signature; Sect. 4.2 then continues to the next column, describing theories over many-sorted empty signatures. Some build on the theories of the previous column, but some are also new. Section 4.3 describes the next column, one-sorted theories over a non-empty signature. Here, we use two constructions to generate new theories from previously introduced ones. One construction adds a function symbol to an empty signature (in a way that preserves all properties), and the second preserves all properties but convexity, making it possible to construct non-convex examples in a uniform way. We also present new theories when the constructions are not sufficient. Finally, Sect. 4.4 describes theories over non-empty many-sorted signatures.[3]

[3] Proofs that each theory has the claimed properties can be found in the appendix to [13].

Table 1. Summary of all possible combinations of theory properties. Shaded cells represent impossible combinations. In line 26: $n > 1$; in line 28: $m > 1$, $n > 1$ and $|m - n| > 1$.

					Empty		Non-empty		
SI	**SM**	**FW**	**SW**	**CV**	One-sorted	Many-sorted	One-sorted	Many-sorted	$N^{\underline{o}}$
T	T	T	T	T	$\mathcal{T}_{\geq n}$	$(\mathcal{T}_{\geq n})^2$	$(\mathcal{T}_{\geq n})_s$	$((\mathcal{T}_{\geq n})^2)_s$	1
				F	Theorem 5		$(\mathcal{T}_{\geq n})_\vee$	$((\mathcal{T}_{\geq n})^2)_\vee$	2
			F	T	Theorem 3	$\mathcal{T}_{2,3}$	$\mathcal{T}_f$	$(\mathcal{T}_f)_s$	3
				F		Theorem 5	$\mathcal{T}_f^s$	$(\mathcal{T}_{2,3})_\vee$	4
		F	T	T	Theorem 2				5
				F					6
			F	T	$\mathcal{T}_\infty$	$(\mathcal{T}_\infty)^2$	$(\mathcal{T}_\infty)_s$	$((\mathcal{T}_\infty)^2)_s$	7
				F	Theorem 5		$(\mathcal{T}_\infty)_\vee$	$((\mathcal{T}_\infty)^2)_\vee$	8
	F	T	T	T	Theorem 7	Unicorn	Theorem 7	Unicorn	9
				F		Theorem 5			10
			F	T	$\mathcal{T}_{even}^\infty$	$(\mathcal{T}_{even}^\infty)^2$	$(\mathcal{T}_{even}^\infty)_s$	$((\mathcal{T}_{even}^\infty)^2)_s$	11
				F	Theorem 5		$(\mathcal{T}_{even}^\infty)_\vee$	$((\mathcal{T}_{even}^\infty)^2)_\vee$	12
		F	T	T	Theorem 2				13
				F					14
			F	T	$\mathcal{T}_{n,\infty}$	$(\mathcal{T}_{n,\infty})^2$	$(\mathcal{T}_{n,\infty})_s$	$((\mathcal{T}_{n,\infty})^2)_s$	15
				F	Theorem 5		$(\mathcal{T}_{n,\infty})_\vee$	$((\mathcal{T}_{n,\infty})^2)_\vee$	16
F	T	T	T	T	Theorem 1				17
				F					18
			F	T					19
				F					20
		F	T	T	Theorems 1 and 2				21
				F					22
			F	T	Theorem 1				23
				F					24
	F	T	T	T	$\mathcal{T}_{\leq 1}$	$(\mathcal{T}_{\leq 1})^2$	$(\mathcal{T}_{\leq 1})_s$	$((\mathcal{T}_{\leq 1})^2)_s$	25
				F	$\mathcal{T}_{\leq n}$	$(\mathcal{T}_{\leq n})^2$	$(\mathcal{T}_{\leq n})_s$	$((\mathcal{T}_{\leq n})^2)_s$	26
			F	T	Theorem 8	$\mathcal{T}_1^{odd}$	$\mathcal{T}_{odd}^{\neq}$	$(\mathcal{T}_1^{odd})_s$	27
				F	$\mathcal{T}_{\langle m,n \rangle}$	$(\mathcal{T}_{\langle m,n \rangle})^2$	$(\mathcal{T}_{\langle m,n \rangle})_s$	$((\mathcal{T}_{\langle m,n \rangle})^2)_s$	28
		F	T	T	Theorem 2				29
				F					30
			F	T	Theorem 6	$\mathcal{T}_1^\infty$	$\mathcal{T}_{1,\infty}^{\neq}$	$(\mathcal{T}_1^\infty)_s$	31
				F		$\mathcal{T}_2^\infty$	$\mathcal{T}_{2,\infty}^{\neq}$	$(\mathcal{T}_2^\infty)_s$	32

4.1 Theories over the One-Sorted Empty Signature

Table 2. Σ_1-theories

Name	Axiomatization
$\mathcal{T}_{\geq n}$	$\{\psi_{\geq n}\}$
$\mathcal{T}_\infty$	$\{\psi_{\geq k} : k \in \mathbb{N}\}$
$\mathcal{T}_{even}^\infty$	$\{\neg\psi_{=2k+1} : k \in \mathbb{N}\}$
$\mathcal{T}_{n,\infty}$	$\{\psi_{=n} \vee \psi_{\geq k} : k \in \mathbb{N}\}$
$\mathcal{T}_{\leq n}$	$\{\psi_{\leq n}\}$
$\mathcal{T}_{\langle m,n \rangle}$	$\{\psi_{=m} \vee \psi_{=n}\}$

Table 3. Σ_2-theories

Name	Axiomatization
$\mathcal{T}_{2,3}$	$\{(\psi_{=2}^{\sigma} \wedge \psi_{\geq k}^{\sigma_2}) \vee (\psi_{\geq 3}^{\sigma} \wedge \psi_{\geq 3}^{\sigma_2}) : k \in \mathbb{N}\}$
$\mathcal{T}_1^{odd}$	$\{\psi_{=1}^{\sigma}\} \cup \{\neg\psi_{=2k}^{\sigma_2} : k \in \mathbb{N}\}$
$\mathcal{T}_1^\infty$	$\{\psi_{=1}^{\sigma}\} \cup \{\psi_{\geq k}^{\sigma_2} : k \in \mathbb{N}\}$
$\mathcal{T}_2^\infty$	$\{\psi_{=2}^{\sigma}\} \cup \{\psi_{\geq k}^{\sigma_2} : k \in \mathbb{N}\}$

The axiomatizations for theories over the one-sorted empty signature Σ_1 are given in Table 2. We briefly describe them here.

For each $n > 0$, $\mathcal{T}_{\geq n}$ includes all structures with domains of cardinality at least n; $\mathcal{T}_{\infty}$ is the theory including all structures whose domains are infinite; $\mathcal{T}_{even}^{\infty}$ has structures with either an even or an infinite number of elements in their domains and was defined in [11], where it was proved to be finitely witnessable, but neither smooth nor strongly finitely witnessable. The proofs justifying Table 1 show additionally that it is stably infinite and convex. $\mathcal{T}_{n,\infty}$ contains those structures whose domains have either exactly n or an infinite number of elements; $\mathcal{T}_{\leq n}$ includes all structures with at most n elements in their domains; and for positive integers m and n, $\mathcal{T}_{\langle m,n\rangle}$ has structures whose domains have either precisely m elements, or precisely n elements. This completes the first column of theory examples.

Example 1. The theory $\mathcal{T}_{\geq n}$ admits all considered properties, while $\mathcal{T}_{\langle m,n\rangle}$ admits only finite witnessability.

4.2 Theories over the Two-Sorted Empty Signature

We next introduce the theories over empty two-sorted signatures. For many cases, we can simply add a trivial sort to one of the theories defined in Sect. 4.1. When this is not possible, we introduce new theories.

Adding a Sort to a Theory. Any Σ_1-theory can be used to generate a Σ_2-theory simply by adding the sort σ_2 to the signature (without changing the axiomatization). This is formalized as follows:

Definition 4. *Let $\mathcal{T}$ be a Σ_1-theory. $(\mathcal{T})^2$ is the Σ_2-theory axiomatized by $Ax(\mathcal{T})$.*

Lemma 3. *A Σ_1-theory $\mathcal{T}$ is stably infinite, smooth, finitely witnessable, strongly finitely witnessable, or convex w.r.t. $\{\sigma\}$ if and only if $(\mathcal{T})^2$ is, respectively, stably infinite, smooth, finitely witnessable, strongly finitely witnessable, or convex w.r.t. $\{\sigma, \sigma_2\}$.*

Using Definition 4 and Lemma 3, we can populate many lines in the second column of examples by extending the corresponding theory from the previous column.

Example 2. $(\mathcal{T}_{\geq n})^2$ is a theory over two sorts, σ and σ_2, whose structures must have at least n elements in the domain of σ (but have no restrictions on the size of the domain of σ_2). As seen in the first line of Table 1, $\mathcal{T}_{\geq n}$ admits all the considered properties. By Lemma 3, so does $(\mathcal{T}_{\geq n})^2$.

Additional Theories over $\boldsymbol{\Sigma_2}$. On some lines, e.g., line 3, there is no Σ_1-theory to extend. In such cases, we cannot use Definition 4 to construct a many-sorted variant.

We introduce the theories shown in Table 3 to cover these cases. The theory $\mathcal{T}_{2,3}$ contains two kinds of structures: (i) structures whose domains both have at least 3 elements; and (ii) structures with exactly two elements in the domain of σ and an infinite number of elements in the domain of σ_2. The theory $\mathcal{T}_1^{odd}$ has structures with exactly one element in the domain of σ and either an odd or an infinite number of elements in the domain of σ_2. The theory $\mathcal{T}_1^{\infty}$ is similar: it has structures with exactly one element in the domain of σ and an infinite number of elements in the domain of σ_2. Finally, $\mathcal{T}_2^{\infty}$ is similar to $\mathcal{T}_1^{\infty}$ except that its structures have exactly 2 elements in the domain of σ.

Example 3. The theory $\mathcal{T}_{2,3}$ was first defined in [4] and later used in [11], where it was proved to be polite (and therefore smooth, stably infinite, and finitely witnessable) without being strongly polite (and therefore not strongly finitely witnessable). The justification proofs for Table 1 show that $\mathcal{T}_{2,3}$ is convex as well.[4]

4.3 Theories over a One-Sorted Non-empty Signature

We continue to the next column, with one-sorted non-empty signatures. Section 4.3 shows how to construct non-empty theories from one-sorted theories over the empty signature, while preserving all their properties. In Sect. 4.3, we provide a similar construction which generates non-convex theories from the theories in the first column of examples. And in Sect. 4.3, we introduce additional theories not captured by the above constructions. Two of these theories are described in more detail in Sect. 4.3.

Extending a Theory with a Unary Function Symbol While Preserving Properties. Whenever we have a theory over an empty signature, we can construct a variant of it over a non-empty signature by introducing a function symbol and interpreting it as the identity function. This extension preserves all the properties that we consider. This is formalized as follows.

Definition 5. *Let Σ_n be an empty signature with sorts $S = \{\sigma_1, \ldots, \sigma_n\}$, and let $\mathcal{T}$ be a Σ_n-theory. The signature Σ_s^n has sorts S and a single unary function symbol s of arity $\sigma_1 \to \sigma_1$, and $(\mathcal{T})_s$ is the Σ_s^n-theory axiomatized by $Ax(\mathcal{T}) \cup \{\forall\, x.\, [s(x) = x]\}$, where x is a variable of sort σ_1.*

Lemma 4. *For every theory $\mathcal{T}$ over an empty signature Σ_n with sorts $S = \{\sigma_1, \ldots, \sigma_n\}$: $\mathcal{T}$ is stably infinite, smooth, finitely witnessable, strongly finitely witnessable, or convex w.r.t. S if and only if $(\mathcal{T})_s$ is, respectively, stably infinite, smooth, finitely witnessable, strongly finitely witnessable, or convex w.r.t. S.*

We use the operator $(\cdot)_s$ in various places in Table 1 in order to obtain examples in non-empty signatures from existing examples over Σ_1 and Σ_2.

Example 4. $(\mathcal{T}_{\geq n})_s$ is a one-sorted theory, whose structures have at least n elements and interpret the function symbol s as the identity. As seen above, $\mathcal{T}_{\geq n}$ admits all the considered properties. By Lemma 4, so does $(\mathcal{T}_{\geq n})_s$.

[4] We thank Oded Padon for raising the question of whether there exists a theory that is polite and convex, but not strongly polite.

Making a Theory Non-convex. The last general construction that we present aims at taking a theory and creating a non-convex variant of it while preserving the other properties we consider. This can be done with the addition of a single unary function symbol s. To define such a theory, we make use of the formula $\psi_\vee$ from Fig. 3. Intuitively, $\psi_\vee$ states that in an interpretation $\mathcal{A}$ in which it holds, $s^{\mathcal{A}}(s^{\mathcal{A}}(a))$ must equal either $s^{\mathcal{A}}(a)$ or a itself; in other words, either $a = s^{\mathcal{A}}(a) = s^{\mathcal{A}}(s^{\mathcal{A}}(a))$, $a = s^{\mathcal{A}}(s^{\mathcal{A}}(a)) \neq s^{\mathcal{A}}(a)$, or $a \neq s^{\mathcal{A}}(a) = s^{\mathcal{A}}(s^{\mathcal{A}}(a))$, as shown in Fig. 4.

$$\boxed{\psi_\vee = \forall\, x.\ \big[\big(s^2(x) = x\big) \vee \big(s^2(x) = s(x)\big)\big]}$$

Fig. 3. The formula $\psi_\vee$ for non-convex theories.

This is especially useful for defining non-convex theories, since $(s^2(x) = x) \vee (s^2(x) = s(x))$ is valid in the theory, but neither $s^2(x) = x$ nor $s^2(x) = s(x)$ is. Notice, of course, that non-convexity is only possible when there are at least two elements available in the domain – otherwise, all equalities are satisfied.

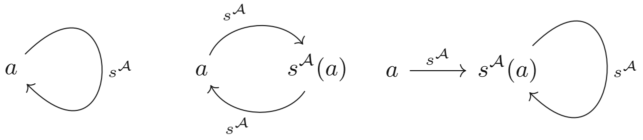


Fig. 4. Possible scenarios when $\psi_\vee$ holds.

Definition 6. *Let $\mathcal{T}$ be a theory over an empty signature with sorts $S = \{\sigma_1, \ldots, \sigma_n\}$. Then $(\mathcal{T})_\vee$ is the Σ_s^n-theory axiomatized by $Ax(\mathcal{T}) \cup \{\psi_\vee\}$.*

Lemma 5. *Let $\mathcal{T}$ be a theory over an empty signature Σ_n with sorts $S = \{\sigma_1, \ldots, \sigma_n\}$. Then: $(\mathcal{T})_\vee$ is stably infinite, smooth, finitely witnessable, or strongly finitely witnessable w.r.t. S if and only if $\mathcal{T}$ is, respectively, stably infinite, smooth, finitely witnessable, or strongly finitely witnessable w.r.t. S. In addition, if $\mathcal{T}$ has a model $\mathcal{A}$ with $|\sigma_1^{\mathcal{A}}| \geq 2$, $(\mathcal{T})_\vee$ is not convex with respect to S.*

Example 5. The theory $(\mathcal{T}_{\geq n})_\vee$ is one-sorted, and its structures have at least n elements. they interpret the symbol s in a way that satisfies $\psi_\vee$. In particular, for each element a of the domain, one of the scenarios from Fig. 4 holds. According to Lemma 5, since $\mathcal{T}_{\geq n}$ admits all properties, $(\mathcal{T}_{\geq n})_\vee$ admits all properties but convexity.

Additional Theories over Σ_s. Whenever there is a Σ_1-theory with some properties, we can obtain a Σ_s theory with the same properties using one of the techniques above. To cover cases for which there is no corresponding Σ_1-theory, we use the theories presented in Table 4 and described below.

Table 4. Σ_s-theories

Name	Axiomatization
$\mathcal{T}_f$	$\{[\psi^{=}_{\geq f_1(k)} \wedge \psi^{\neq}_{\geq f_0(k)}] \vee \bigvee_{i=1}^{k}[\psi^{=}_{=f_1(i)} \wedge \psi^{\neq}_{=f_0(i)}] : k \in \mathbb{N} \setminus \{0\}\}$
$\mathcal{T}_f^s$	$Ax(\mathcal{T}_f) \cup \{\psi_{\vee}\}$
$\mathcal{T}^{\neq}_{odd}$	$\{\psi_{=1} \vee [\neg\psi_{=2k} \wedge \forall\, x.\ \neg(s(x) = x)] : k \in \mathbb{N}\}$
$\mathcal{T}^{\neq}_{1,\infty}$	$\{\psi_{=1} \vee [\psi_{\geq k} \wedge \forall\, x.\ \neg(s(x) = s)] : k \in \mathbb{N}\}$
$\mathcal{T}^{\neq}_{2,\infty}$	$\{[\psi_{=2} \wedge \forall\, x.\ (s(x) = x)] \vee [\psi_{\geq k} \wedge \forall\, x.\ \neg(s(x) = x)] : k \in \mathbb{N}\}$

We start with $\mathcal{T}^{\neq}_{odd}$, $\mathcal{T}^{\neq}_{1,\infty}$, and $\mathcal{T}^{\neq}_{2,\infty}$, deferring the discussion on $\mathcal{T}_f$ and $\mathcal{T}_f^s$ to Sect. 4.3. The theory $\mathcal{T}^{\neq}_{odd}$ has structures $\mathcal{A}$ with either an infinite or an odd number of elements and with the property that if $\mathcal{A}$ is not trivial, then $s^{\mathcal{A}}(a) \neq a$ for all $a \in \sigma^{\mathcal{A}}$. The theory $\mathcal{T}^{\neq}_{1,\infty}$ has all structures $\mathcal{A}$ that either: (i) are trivial; or (ii) have infinitely many elements and for which $s^{\mathcal{A}}(a) \neq a$ for each $a \in \sigma^{\mathcal{A}}$. Similarly, $\mathcal{T}^{\neq}_{2,\infty}$ has structures $\mathcal{A}$ that either: (i) have exactly two elements and interpret s as the identity; or (ii) have infinitely many elements and interpret s in such a way that $s^{\mathcal{A}}(a) \neq a$ for all $a \in \sigma^{\mathcal{A}}$.

On the Theories $\mathcal{T}_f$ and $\mathcal{T}_f^s$. We now introduce the theories $\mathcal{T}_f$ and $\mathcal{T}_f^s$. The importance of these theories is that both of them are *one-sorted* theories that are polite but not strongly polite (the first is also convex and the second is not). Their existence improves on the result of [11], which introduced a *two-sorted* theory that is polite but not strongly polite (namely $\mathcal{T}_{2,3}$).

For their axiomatizations, we use the formulas from Fig. 5, in which s is a unary function symbol. $\psi^{=}_{\geq n}$ ($\psi^{=}_{=n}$) states that a structure $\mathcal{A}$ has at least (exactly) n elements a satisfying $s^{\mathcal{A}}(a) = a$; similarly, $\psi^{\neq}_{\geq n}$ ($\psi^{\neq}_{=n}$) states that a structure $\mathcal{A}$ has at least (exactly) n elements a satisfying $s^{\mathcal{A}}(a) \neq a$.

Further, the axiomatization requires a function f from positive integers to $\{0, 1\}$ that is not computable with the property that for $k > 0$, f maps half of the numbers in the interval $[1, 2^k]$ to 1 and the other half to 0. The existence of such a function is formalized below. We start by defining counting functions f_0 and f_1.

Definition 7. *Let* $f : \mathbb{N} \setminus \{0\} \to \{0, 1\}$*. For* $i \in \{0, 1\}$ *and* $n \in \mathbb{N}$*,* $f_i(n)$ *is defined by:* $f_i(n) = |f^{-1}(i) \cap [1, n]|$*.*

Intuitively, $f_0(n)$ counts how many numbers between 1 and n (inclusive) are mapped by f to 0 and $f_1(n)$ counts how many are mapped to 1. Because $f(n)$ always equals 0 or 1, it is easy to see that for every $n > 0$, $n = f_1(n) + f_0(n)$.

$$\psi^{=}_{\geq n} = \exists\, \overrightarrow{x}.\, [\bigwedge_{i=1}^{n} p(x_i) \wedge \delta_n], \qquad \psi^{\neq}_{\geq n} = \exists\, \overrightarrow{x}.\, [\bigwedge_{i=1}^{n} \neg p(x_i) \wedge \delta_n],$$

$$\psi^{=}_{= n} = \exists\, \overrightarrow{x}.\, [\delta_n \wedge \bigwedge_{i=1}^{n} p(x_i) \wedge \forall\, x.\, [p(x) \rightarrow \bigvee_{i=1}^{n} x = x_i]],$$

$$\psi^{\neq}_{= n} = \exists\, \overrightarrow{x}.\, [\delta_n \wedge \bigwedge_{i=1}^{n} \neg p(x_i) \wedge \forall\, x.\, [\neg p(x) \rightarrow \bigvee_{i=1}^{n} x = x_i]].$$

Fig. 5. Cardinality formulas for signatures with a unary function symbol s. $\overrightarrow{x}$ stands for $x_1, \ldots, x_n$, $p(x)$ for $s(x) = x$, and δ_n for $\bigwedge_{1 \leq i < j \leq n} \neg(x_i = x_j)$.

Lemma 6. *There exists a function* $f : \mathbb{N} \setminus \{0\} \rightarrow \{0, 1\}$ *such that* $f(1) = 1$ *with the properties that:* f *is not computable; and, for every* $k \in \mathbb{N} \setminus \{0\}$, $f_0(2^k) = f_1(2^k)$.

Example 6. The constant function that assigns 0 to all positive integers satisfies neither the first nor the second condition of Lemma 6. The function that assigns 0 to even numbers and 1 to odd numbers satisfies the second condition, but not the first. Of course, any non-computable function satisfies the first condition. An example could be found by a function that returns 1 if the Turing machine that is encoded by the given number halts and 0 otherwise, under some encoding. Finding a function that admits both conditions is more challenging.

Let f be some function with the properties listed in Lemma 6. We can now define $\mathcal{T}_f$ over Σ_s (note that f itself is not a part of the signature, but is rather used to help define the axioms of $\mathcal{T}_f$). $\mathcal{T}_f$ consists of those structures $\mathcal{A}$ that either (i) have a finite cardinality n, with $f_1(n)$ elements satisfying $s^{\mathcal{A}}(a) = a$, and $f_0(n)$ elements satisfying $s^{\mathcal{A}}(a) \neq a$ (and thus $\mathcal{A}$ satisfies $\psi^{=}_{\geq f_1(k)} \wedge \psi^{\neq}_{\geq f_0(k)}$ for $k \leq n$, and $\psi^{=}_{= f_1(n)} \wedge \psi^{\neq}_{= f_0(n)}$ and hence $\bigvee_{i=1}^{k} [\psi^{=}_{= f_1(i)} \wedge \psi^{\neq}_{= f_0(i)}]$ for all $k \geq n$); or (ii) have infinitely many elements, with infinitely many elements satisfying each condition, $s^{\mathcal{A}}(a) = a$ and $s^{\mathcal{A}}(a) \neq a$ (and thus $\mathcal{A}$ satisfies $\psi^{=}_{\geq f_1(k)} \wedge \psi^{\neq}_{\geq f_0(k)}$ for all $k \in \mathbb{N}$). Note that the description is well-defined because an element must always satisfy either $s^{\mathcal{A}}(a) = a$ or $s^{\mathcal{A}}(a) \neq a$, but never both or neither of these. The theory $\mathcal{T}^s_f$ is similar to $\mathcal{T}_f$, but in addition to $Ax(\mathcal{T}_f)$ its structures must also satisfy $\psi_{\vee}$.

Remark 1. The construction of $\mathcal{T}^s_f$ from $\mathcal{T}_f$ is very similar to the general construction of Definition 6. However, the corresponding result, Lemma 5, according to which all properties but convexity are preserved by this operation, is only shown in Lemma 5 for cases where the original signature is empty, which is not the case for $\mathcal{T}_f$. Obtaining $\mathcal{T}^s_f$ from $\mathcal{T}_f$ is not done by adding a function symbol, but rather by changing the axiomatization of the already existing function symbol. While we do prove that $\mathcal{T}^s_f$ has the required properties, a general result in the style of Lemma 5 for arbitrary signatures, with the ability to preserve an existing function symbol instead of adding a new one, is left for future work.

Example 7. Let $\mathcal{A}_n$ be a Σ_s-model with domain $\{a_1, \ldots, a_n\}$ such that: $s^{\mathcal{A}_n}(a_i)$ equals a_i if $1 \leq i \leq f_1(n)$, and a_1 if $f_1(n) < i \leq n$ (the second condition may be void if $n = 1$). Then $\mathcal{A}_n$ is a model of both $\mathcal{T}_f$ and $\mathcal{T}_f^s$.

If κ is an infinite cardinal, let $\mathcal{A}_\kappa$ be a Σ_s-model with domain $A \cup \{a_n : n \in \mathbb{N} \setminus \{0\}\}$ (where A is a set of cardinality κ disjoint from $\{a_n : n \in \mathbb{N} \setminus \{0\}\}$) such that $s^{\mathcal{A}_\kappa}(a_i) = a_i$ for each $i \in \mathbb{N} \setminus \{0\}$, and $s^{\mathcal{A}_\kappa}(a) = a_1$ for each $a \in A$. Then $\mathcal{A}_\kappa$ is a model of both $\mathcal{T}_f$ and $\mathcal{T}_f^s$.

To show that $\mathcal{T}_f$ is smooth and finitely witnessable, we construct, given a $\mathcal{T}_f$-interpretation. another $\mathcal{T}_f$-interpretation by (possibly) adding two disjoint sets of elements to the interpretation, one whose elements will satisfy $s(a) = a$, and one whose elements will satisfy $s(a) \neq a$.

To show that it is not strongly finitely witnessable, we use the following lemmas, which are interesting in their own right. According to the first, the *mincard* function of $\mathcal{T}_f$ is not computable.

Lemma 7. *The mincard function of $\mathcal{T}_f$ is not computable.*

The second lemma that is needed in order to prove that $\mathcal{T}_f$ is not strongly finitely witnessable, is quite surprising. As it turns out, for quantifier-free formulas, the set of $\mathcal{T}_f$-satisfiable formulas coincides with the set of satisfiable formulas. That is, even though the definition of $\mathcal{T}_f$ is very complex, it induces the same satisfiability relation, over quantifier-free formulas, as the simplest theory possible – the theory axiomatized by the empty set (or, equivalently, all valid first-order sentences).

Lemma 8. *Every quantifier-free Σ_s-formula that is satisfiable is $\mathcal{T}_f$-satisfiable.*

Note that Lemma 8 does not hold for quantified formulas in general. For example, the formula $\forall x.\ s(x) \neq x$ is satisfiable but not $\mathcal{T}_f$-satisfiable: because $f(1) = 1$, every $\mathcal{T}_f$-interpretation $\mathcal{A}$ must have at least one element a with $s^{\mathcal{A}}(a) = a$.

Using Lemma 7 and 8, it is possible to show that $\mathcal{T}_f$ is not strongly finitely witnessable:

Lemma 9. *$\mathcal{T}_f$ is not strongly finitely witnessable.*

The idea of the proof of Lemma 9 goes as follows: assume for contradiction that there is a strong witness *wit*. The *mincard* function for $\mathcal{T}_f$ can then be defined as

$$mincard(\phi) = \min\{|V/E| : E \in eq \,\text{and}\, wit(\phi) \wedge \delta_V^E \,\text{is}\, \mathcal{T}_f\text{-satisfiable}\}, \qquad (1)$$

where eq is the set of all equivalence relations E on $V = vars(wit(\phi))$, being the corresponding arrangements denoted by δ_V^E. Clearly, the sets V and eq can be effectively computed. Also, by Lemma 8, testing for the $\mathcal{T}_f$-satisfiability of quantifier-free formulas is decidable. Together with our assumption that *wit* is computable, we get that the *mincard* function of $\mathcal{T}_f$ is computable, which contradicts Lemma 7.

The arguments for $\mathcal{T}_f^s$ are very similar, and require minor changes in the corresponding proofs for $\mathcal{T}_f$.

Remark 2. We remark on the connection between the results regarding $\mathcal{T}_f$ and $\mathcal{T}_f^s$, and those of [3]. What we show here is that $\mathcal{T}_f$ ($\mathcal{T}_f^s$) is polite but not strongly polite. Figure 1 of [3] summarizes the relations between these two properties for the one-sorted case. It shows that polite theories that are axiomatized by a universal set of axioms, and whose quantifier-free satisfiability problem is decidable, are strongly polite. While $\mathcal{T}_f$ is decidable for quantifier-free formulas (this is a corollary of Lemma 8), its presentation here is definitely not as a universal theory. On the other hand, [3] also shows that decidable polite theories for which checking if a finite interpretation belongs to the theory is decidable are also strongly polite. However, it is undecidable, given an interpretation, to check whether it belongs to $\mathcal{T}_f$ (and $\mathcal{T}_f^s$): such an algorithm would lead to an algorithm to compute f as well. Thus, the theories $\mathcal{T}_f$ and $\mathcal{T}_f^s$ are polite, but do not meet the criteria for strong politeness from [3]. And indeed, they are not strongly polite.

4.4 Theories over Many-Sorted Non-empty Signatures

For the last column of Table 1, all possible theories can be obtained from theories that were already defined, using a combination of Definitions 4 to 6, and so there is no need to present additional theories specifically for many-sorted non-empty signatures.

Example 8. Line 1 includes the theory $((\mathcal{T}_{\geq n})^2)_s$, obtained from $(\mathcal{T}_{\geq n})^2$ using Definition 5, where the latter theory is obtained from $\mathcal{T}_{\geq n}$ using Definition 4. This theory admits all properties, including convexity. To obtain a non-convex variant, the theory $((\mathcal{T}_{\geq n})^2)_\vee$ is constructed in a similar fashion, using Definition 6 instead of Definition 5.

With many-sorted non-empty signatures, we can always find an example for each combination of properties, except for those that are trivially impossible due to Theorems 1 and 2 (i.e., theories that are strongly finitely witnessable but not finitely witnessable and theories that are smooth but not stably infinite). This is nicely depicted by Fig. 6. Theorems 1 and 2 are represented in this figure by the location of the circles: the circle for smooth theories is entirely inside the circle for stably infinite theories, and similarly for strongly finitely witnessable and finitely witnessable theories. Then, for every region in this figure, the rightmost column of Table 1 has an example, the sole exception being the region that represents unicorn theories.

Remark 3. For non-empty signatures, we chose to include functions rather than predicates. This is not essential as we can replace function symbols by predicate symbols by including the sort of the result of the function as the last component of the arity of the predicate, and then adding an axiom that forces the predicate to be a function.

5 Polite Combination Without Smoothness

Polite combination of theories was introduced in [10]. There, it was claimed that in order to combine a theory $\mathcal{T}$ with any other theory using polite combination, it suffices for $\mathcal{T}$ to be smooth and finitely witnessable (that is, polite). Later, in [6], this condition was corrected, and it was shown that in fact a stronger requirement is needed from $\mathcal{T}$: it has to be smooth and strongly finitely witnessable (that is, strongly polite) to be applicable for the combination method.

Given that weakening strong finite witnessability to finite witnessability results in a condition that does not suffice, it is natural to ask whether there is any other way to weaken the required conditions for polite combination. Rather than weakening strong finite witnessability to finite witnessability, here we consider another option: weakening the smoothness condition to stable infiniteness. Thus, the main result of this section is that polite combination can be done for theories that are stably infinite and strongly finitely witnessable, even if they are not smooth.

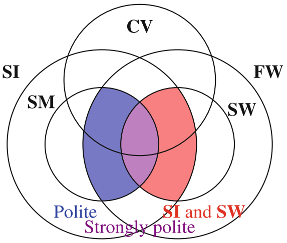


Fig. 6. A diagram of the various notions studied in this paper. (Color figure online)

Our contribution can be understood by viewing Fig. 6, ignoring the circle that represents convexity (a property unrelated to the current section). [6] shows that polite combination can be done for the purple region, which represents smooth and strongly finitely witnessable theories. [6] also presented an example showing that expanding the same combination method to the blue region, which represents smooth and finitely witnessable theories, results in an error. Here we instead expand polite combination to the red region, which represents stably infinite and strongly finitely witnessable theories. Now, the red region, if not empty, is only populated by unicorn theories (see Sect. 4). If such theories do not exist, the result follows immediately. Until this is settled, however, we provide a direct proof, regardless of the existence of unicorn theories.

The next theorem shows that polite theory combination can be done for theories that are not necessarily strongly polite (smooth and strongly finitely witnessable), but rather that are simply stably infinite and strongly finitely witnessable.

Theorem 9. *Let Σ_1 and Σ_2 be disjoint signatures with sorts S_1 and S_2; let $\mathcal{T}_1$ be a Σ_1-theory, $\mathcal{T}_2$ be a Σ_2-theory, and $\mathcal{T} = \mathcal{T}_1 \oplus \mathcal{T}_2$; and let ϕ_1 be a quantifier-free Σ_1-formula and ϕ_2 a quantifier-free Σ_2-formula.*

Assume that $\mathcal{T}_2$ is stably-infinite and strongly finitely witnessable w.r.t. $S = S_1 \cap S_2$, with strong witness wit. Let $\psi = wit(\phi_2)$, $V_\sigma = vars_\sigma(\psi)$ for every $\sigma \in S$ and $V = \bigcup_{\sigma \in S} vars_\sigma(\psi)$. Then the following are equivalent:

1. *$\phi_1 \wedge \phi_2$ is $\mathcal{T}$-satisfiable;*
2. *there exists an arrangement δ_V over V such that $\phi_1 \wedge \delta_V$ is $\mathcal{T}_1$-satisfiable and $\psi \wedge \delta_V$ is $\mathcal{T}_2$-satisfiable.*

It relies heavily on the following lemma, that proves that stable infiniteness and strong finite witnessability imply a weaker notion of smoothness. In this weaker notion, uncountable domains in the original structure $\mathcal{A}$ are reduced to countable ones, and the function κ, that dictates the cardinalities of models, is assumed to never assign an uncountable cardinal to any of the sorts.

Lemma 10. *Let Σ be a signature with $S \subseteq \mathcal{S}_\Sigma$, and $\mathcal{T}$ a theory over Σ. If $\mathcal{T}$ is a stably-infinite and strongly finitely witnessable theory, both w.r.t. the set of sorts S, then: for every quantifier-free Σ-formula ϕ; $\mathcal{T}$-interpretation $\mathcal{A}$ that satisfies ϕ; and function κ from $S_\omega^{\mathcal{A}} = \{\sigma \in S : |\sigma^{\mathcal{A}}| \leq \omega\}$ to the class of cardinals such that $|\sigma^{\mathcal{A}}| \leq \kappa(\sigma) \leq \omega$ for every $\sigma \in S_\omega^{\mathcal{A}}$, there exists a $\mathcal{T}$-interpretation $\mathcal{B}$ that satisfies ϕ with $|\sigma^{\mathcal{B}}| = \kappa(\sigma)$ for every $\sigma \in S_\omega^{\mathcal{A}}$, and $|\sigma^{\mathcal{B}}| = \omega$ for every $\sigma \in S \setminus S_\omega^{\mathcal{A}}$.*

The proof of Theorem 9 goes as follows: first, we make the infinite domains corresponding to shared sorts of a model $\mathcal{A}$ of $\phi_1 \wedge \delta_V$ at most countable, by applying Lemma 2. We then proceed similarly to the proof of the polite combination method in [6]: decrease a model $\mathcal{B}$ of $\psi \wedge \delta_B$ by using *wit* as a strong witness; and then make the cardinalities of the shared sorts in $\mathcal{B}$ equal those of $\mathcal{A}$ (which are at most countable), by using Lemma 10.

This result greatly improves the state-of-the-art in polite theory combination, which requires proving that one of the theories is both smooth and strongly finitely witnessable. Thanks to this theorem, proving smoothness can be replaced by proving stable infiniteness, which is typically a much easier task.

6 Conclusion

As mentioned, there are two main contributions offered in this paper, both associated with the theme of theory combination. In Sect. 4, we provide a table with examples for almost all the combinations of stable infiniteness, smoothness, convexity, finite witnessability, and strong finite witnessability known not to be impossible. Section 3 provides theorems proving the sharpness of the examples provided. The second contribution is a new combination theorem, according to which polite theory combination can be done without smoothness, provided we have instead stable infiniteness.

Many ideas for future work rise from the studies here presented. A first direction would be to settle the question of whether unicorn theories exist: if they do not, a proof would probably involve an interesting generalization of the upward Löwenheim-Skolem theorem for many-sorted logic and would imply that strongly polite theories are just simply stably-infinite and strongly finitely witnessable theories, thus greatly simplifying the proof of Theorem 9; if unicorn theories do exist, one wonders if they can be combined in some meaningful way. Another direction of future work involves considering other model-theoretic properties in our table, such as shininess, gentleness, flexibility, and so on, as well as the effect of taking proper subsets of sorts for signatures containing more than one sort.

Acknowledgments. This work was funded in part by NSF-BSF grant numbers 2110397 (NSF) and 2020704 (BSF) and ISF grant number 619/21.

References

1. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
2. Barrett, C.W., Dill, D.L., Stump, A.: A generalization of Shostak's method for combining decision procedures. In: Armando, A. (ed.) FroCoS 2002. LNCS (LNAI), vol. 2309, pp. 132–146. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45988-X_11
3. Casal, F., Rasga, J.: Revisiting the equivalence of shininess and politeness. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 198–212. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_15
4. Casal, F., Rasga, J.: Many-sorted equivalence of shiny and strongly polite theories. J. Autom. Reason. **60**(2), 221–236 (2018). https://doi.org/10.1007/s10817-017-9411-y
5. Fontaine, P.: Combinations of theories for decidable fragments of first-order logic. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS (LNAI), vol. 5749, pp. 263–278. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04222-5_16
6. Jovanović, D., Barrett, C.: Polite theories revisited. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR 2010. LNCS, vol. 6397, pp. 402–416. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16242-8_29
7. Krstić, S., Goel, A., Grundy, J., Tinelli, C.: Combined satisfiability modulo parametric theories. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 602–617. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_47
8. Monzano, M.: Introduction to many-sorted logic. In: Meinke, K., Tucker, J.V. (eds.) Many-sorted Logic and its Applications. Wiley Professional Computing, Wiley, Hoboken (1993)
9. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. **1**(2), 245–257 (1979). https://doi.org/10.1145/357073.357079
10. Ranise, S., Ringeissen, C., Zarba, C.G.: Combining data structures with nonstably infinite theories using many-sorted logic. In: Gramlich, B. (ed.) FroCoS 2005.

LNCS (LNAI), vol. 3717, pp. 48–64. Springer, Heidelberg (2005). https://doi.org/10.1007/11559306_3. https://hal.inria.fr/inria-00000570

11. Sheng, Y., Zohar, Y., Ringeissen, C., Reynolds, A., Barrett, C., Tinelli, C.: Politeness and stable infiniteness: stronger together. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 148–165. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_9
12. Tinelli, C., Zarba, C.: Combining decision procedures for theories in sorted logics. Technical report 04-01, Department of Computer Science, The University of Iowa (2004). https://doi.org/10.1007/978-3-540-30227-8_53
13. Toledo, G.V., Zohar, Y., Barrett, C.: Combining combination properties: an analysis of stable infiniteness, convexity, and politeness. arXiv arXiv:2305.02384 (2023). https://arxiv.org/abs/2305.02384. Accepted to CADE 2023

Decidability of Difference Logic over the Reals with Uninterpreted Unary Predicates

Bernard Boigelot, Pascal Fontaine, and Baptiste Vergain(✉)

Montefiore Institute, B28, Université de Liège, Liège, Belgium
{Bernard.Boigelot,Pascal.Fontaine,BVergain}@uliege.be

Abstract. First-order logic fragments mixing quantifiers, arithmetic, and uninterpreted predicates are often undecidable, as is, for instance, Presburger arithmetic extended with a single uninterpreted unary predicate. In the SMT world, difference logic is a quite popular fragment of linear arithmetic which is less expressive than Presburger arithmetic. Difference logic on integers with uninterpreted unary predicates is known to be decidable, even in the presence of quantifiers. We here show that (quantified) difference logic on real numbers with a single uninterpreted unary predicate is undecidable, quite surprisingly. Moreover, we prove that difference logic on integers, together with order on reals, combined with uninterpreted unary predicates, remains decidable.

Keywords: First-order logic · Decidability · SMT · Arithmetic · Uninterpreted predicates

1 Introduction

The success of satisfiability modulo theories (SMT) solvers in verification can be attributed to several things, but one of them is indisputably the omnipresence, in the combination of theories, of arithmetic reasoners. As SMT solvers get stronger in quantified reasoning, it becomes more interesting to get a clear picture of decidability frontiers when arithmetic is used in a quantified SMT context. Some pure arithmetic theories are already undecidable, even in their quantifier-free fragment, e.g., Peano arithmetic [12], i.e., a first-order theory of the natural numbers with addition and multiplication. However, Presburger arithmetic, somehow the linear restriction of Peano arithmetic, is decidable even in the quantified case [10], but augmenting Presburger arithmetic with a single unary uninterpreted predicate already yields undecidability [7,11,19]. To obtain a decidable fragment mixing arithmetic and uninterpreted predicates, one must further restrict the expressiveness.

In the SMT world, difference logic used to be a popular fragment of arithmetic, because of its low complexity in the quantifier-free case. In this fragment, arithmetic is limited to difference constraints of the form $x - y \bowtie c$ where x

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 542–559, 2023.
https://doi.org/10.1007/978-3-031-38499-8_31

and y are variables, c is an integer constant and $\bowtie$ belongs to $\{<, \leq, =, \geq, >\}$. Difference constraints can, e.g., express conditions on the distance between two variables, the atomic formula $x - y = 2$ stating that the distance between the values of x and y must be exactly 2. Notice that since difference constraints involve only two variables (c is an integer constant) those constraints are strictly less expressive than linear constraints in Presburger arithmetic. The decidability of the logic mixing difference constraints and unary uninterpreted predicates, when interpreted over $\mathbb{N}$ (or similarly $\mathbb{Z}$) reduces to the decidability of the monadic second-order theory of one successor, usually referred to as S1S. The decidability of S1S has been established thanks to the concept of infinite-word automaton [4].

On the real domain, it is well known that the first-order theory of real-closed fields, which is in a sense the real counterpart of Peano arithmetic, is decidable [20] even in the presence of quantifiers. Whereas this might give the impression that decidability is more often obtained on the reals than on the integers, we here prove that the logic mixing difference constraints and unary uninterpreted predicates, when interpreted over $\mathbb{R}$, is undecidable.

Further restricting the arithmetic language, and considering order on the real domain only, it is known that the monadic second-order theory of order is undecidable [9,17], but its universal fragment is decidable [5]. In this work, we establish that the fragment mixing unary uninterpreted predicates, difference constraints over integer variables, and order constraints over real variables is decidable.

Section 2 provides some prerequisites and the precise definition of the studied fragments. In Sect. 3, we prove the decidability of the fragment mixing unary uninterpreted predicates, difference constraints over integer variables, and order constraints over real variables. This was already the subject of a work-in-progress workshop paper [1]. In Sect. 4, we prove that the fragment of quantified difference constraints over real variables extended with a single unary uninterpreted predicate is undecidable.

2 Preliminaries

We refer to e.g., [8] for a general introduction to first-order logic with equality, and assume that the reader is familiar with the notions of signature, term, variable, and formula. We use the usual logical connectives ($\vee$, $\wedge$, $\neg$, $\Rightarrow$, $\Leftrightarrow$) and first-order quantification $\exists x.\, \varphi$ and $\forall x.\, \varphi$, respectively equivalent to writing $\exists x\, (\varphi)$ and $\forall x\, (\varphi)$, i.e., the dot stands for an opening parenthesis that is closed at the end of the formula. Variable symbols are denoted by x, y, z, ... and are meant to be interpreted as real numbers.

Our signature contains the interpreted arithmetic symbols 0, 1, $+$, $-$, $<$, $\leq$, $\geq$, $>$, $=$, and other constants in $\mathbb{N}$ that stand for terms $1 + 1 + \cdots + 1$. We furthermore use a monadic (i.e., unary) interpreted predicate $x \in \mathbb{Z}$ to denote that x has an integer value. The signature also contains uninterpreted predicate symbols P, Q, ... In the whole article, we only consider unary predicate symbols. Indeed, including binary uninterpreted predicates without restriction on first-order quantification directly yields undecidability. Our language is the set of all well-formed formulas, in the usual sense, built using symbols from the signature. Further specific restrictions will be introduced later.

An interpretation specifies a domain (i.e., a set of elements), assigns a value in the domain to each free variable, and assigns relations of appropriate arity on the domain to predicate symbols in the signature. Throughout the article, the interpretation domain is always $\mathbb{R}$. The arithmetic symbols 0, 1, $+$, $-$, $<$, $\leq$, $\geq$, $>$, $=$ are interpreted as expected on $\mathbb{R}$, and $x \in \mathbb{Z}$ is true if and only if x has an integer value[1]. An interpretation assigns an arbitrary subset of the domain $\mathbb{R}$ to each unary predicate. By extension, an interpretation assigns a value in $\mathbb{R}$ to every term, and a truth value to every formula. We denote the interpretation I of a variable x by $I[x]$, and the interpretation of a predicate P by $I[P]$. A model of a formula is an interpretation that assigns true to this formula. A formula is satisfiable on a domain (here $\mathbb{R}$) if it has a model on that domain.

2.1 Difference Arithmetic with Unary Predicates

We consider several fragments where the language is restricted, in particular in the way that the arithmetic relations can be used. A fragment is decidable if there exists a procedure to check whether a given formula in this fragment is satisfiable.

In the various fragments introduced below, all arithmetic atoms are either *order constraints* of the form $x \bowtie y$, or *difference constraints* of the form $x - y \bowtie c$, where x and y are variables, c is a constant in $\mathbb{Z}$, and $\bowtie \in \{<, \leq, =, \geq, >\}$. As a reminder, the language of our formulas only contains *unary predicates*. The only atoms besides the arithmetic ones are of the form $P(x)$ where P is an uninterpreted predicate symbol and x is a variable, and $x \in \mathbb{Z}$ where x is a variable. Note that the addition of constraints of the form $x \bowtie c$, where x is a variable and c is an integer constant, to fragments that already admit difference constraints does not increase their expressive power: constraints $x \bowtie c$ can be replaced by difference constraints $x - v_0 \bowtie c$, where v_0 is a particular variable in $\mathbb{Z}$ intended to be interpreted as zero. Indeed, shifting an interpretation by a fixed integer j — i.e., the new interpretation of any variable x is the old value of x plus j, and the new value of any predicate P for a real number $d + j$ is the old value of P for d — preserves the assigned value of formulas in our fragments. Therefore any model where v_0 is an arbitrary integer can be shifted into a model where v_0 is zero.

As syntactic sugar, conjunctions of order constraints will be merged to improve readability, i.e., we will often write $x < y < z$ rather than $x < y \wedge y < z$. Finally, we use the shorthand $P(x + c)$ instead of $\exists y.\, y - x = c \wedge P(y)$, where x is a free variable and $c \in \mathbb{Z}$.

We now introduce our fragments of interest. Their names are inspired from the SMT-LIB nomenclature, where acronyms stand for the theories that appear in the combinations:

[1] In the current context, this choice of notation for mixed integer-real arithmetic is simpler than using a multi-sorted logic.

- UF1: the theory of uninterpreted functions, with the restriction that uninterpreted symbols may only correspond to monadic predicates;
- RO: the theory of order on the reals only;
- IRO: the theory of order on the reals and integers;
- IDL: difference logic on the integers;
- RDL: difference logic on the reals.

UF1·RO. The fragment UF1·RO is the fragment with unary uninterpreted predicates and order constraints between variables interpreted over $\mathbb{R}$. Difference logic constraints and atoms of the form $x \in \mathbb{Z}$ are not allowed.

Example: The formula $\forall x \, \exists y, z \,.\, y < x < z \wedge \forall t \,.\, (y < t < z \wedge P(t)) \Rightarrow t = x$ describes a predicate P that is true only on isolated real numbers.

UF1·IRO. The fragment UF1·IRO is the extension of UF1·RO where atoms of the form $x \in \mathbb{Z}$ are allowed. This fragment can express order relations between real and integer variables.

Example: The formula $\forall x, y. \, (x < y \wedge x \in \mathbb{Z} \wedge y \in \mathbb{Z}) \Rightarrow \exists v. \, x < v < y \wedge P(v)$ describes a predicate P that is true for at least one value located between any two integers.

UF1·IDL·IRO. The fragment UF1·IDL·IRO is an extension of the fragment UF1·IRO (and therefore of UF1·RO). It is also interpreted over $\mathbb{R}$. Order constraints between variables and atoms of the form $x \in \mathbb{Z}$ are allowed. Additionally, difference logic constraints are allowed, but they can only involve *integer-guarded* variables.

In order to enforce this integer-guard restriction on difference logic constraints, UF1·IDL·IRO formulas must be *well-guarded*, i.e., difference logic constraints can only appear in the two following contexts:

- $x \in \mathbb{Z} \wedge y \in \mathbb{Z} \wedge x - y \bowtie c$,
- $(x \in \mathbb{Z} \wedge y \in \mathbb{Z}) \Rightarrow x - y \bowtie c$,

where x and y are variables, $c \in \mathbb{Z}$ is a constant, and $\bowtie \in \{<, \leq, =, \geq, >\}$.

Example: The following formula describes a predicate that is either true on all odd numbers and false on all even numbers, or the opposite, as well as true on all non-integer numbers:

$$\big[\forall x, y. \, \big(x \in \mathbb{Z} \wedge y \in \mathbb{Z} \wedge y - x = 2\big) \Rightarrow \big(P(x) \Leftrightarrow P(y)\big)\big]$$
$$\wedge \big[\exists x, y. \, x \in \mathbb{Z} \wedge y \in \mathbb{Z} \wedge P(x) \wedge \neg P(y)\big] \wedge \big[\forall z. \, \neg(z \in \mathbb{Z}) \Rightarrow P(z)\big]$$

UF1·RDL. The fragment UF1·RDL is the fragment interpreted over $\mathbb{R}$, where order constraints, difference logic constraints and unary predicate atoms are allowed without any restriction. The use of atoms of the form $x \in \mathbb{Z}$ is forbidden. Since order constraints are a special case of difference logic constraints, the name of the fragment only refers to RDL and not RO.

Example: The formula $\forall x \, \exists y. \, 0 < y - x < 3 \wedge P(y)$ describes a predicate P such that any subinterval of $\mathbb{R}$ of length greater or equal to 3 contains a value for which P is true.

Note: It might appear to the reader that a missing logic in this nomenclature is UF1·IRDL, with difference logic constraints on both real and integer variables. We will later show that UF1·RDL is already undecidable, so it makes little sense to introduce any extension of it.

3 Decidability of UF1·IDL·IRO

The fragment UF1·RO is actually a restriction of the universal fragment of the monadic second-order theory of the real order $\mathbb{R}$, i.e., UF1·RO augmented with universal quantification of predicate variables. It has been established in [5] that the universal fragment of the monadic second-order theory of the real order $\mathbb{R}$ is decidable, which trivially implies the decidability of UF1·RO. We show here that its extension UF1·IDL·IRO (and therefore UF1·IRO) is also decidable, by a reduction to UF1·RO.

Theorem 1. UF1·IDL·IRO *and* UF1·IRO *are decidable.*

Note that the decidability of UF1·IRO is a direct consequence of the decidability of UF1·IDL·IRO, since UF1·IDL·IRO is an extension of UF1·IRO. The remaining of this section is thus dedicated to proving that UF1·IDL·IRO is decidable.

3.1 Recognizing Integer Values

We first show how to define in UF1·RO a predicate P_{int} over $\mathbb{R}$ that is $<$-isomorphic to $\mathbb{Z}$, i.e., such that there exists a bijection between the sets described by P_{int} and $\mathbb{Z}$ that preserves the order relation over their elements. Integer guards in UF1·IDL·IRO will later be translated using P_{int}. Intuitively, an integer-guarded variable in a UF1·IDL·IRO formula will correspond to a variable taking its value in the set described by P_{int} in the translated UF1·RO formula.

We axiomatize P_{int} in UF1·RO as follows:

- Every element of P_{int} is isolated:
  $\forall x\, \exists\, y, z.\ y < x < z \wedge \forall t.\ [y < t < z \wedge P_{int}(t)] \Rightarrow t = x.$
- Every point in $\mathbb{R}$ has a unique successor in P_{int}:
  $\forall x\, \exists\, y.\ x < y \wedge P_{int}(y) \wedge \forall t.\ x < t < y \Rightarrow \neg P_{int}(t).$
- Similarly, every point in $\mathbb{R}$ has a unique predecessor in P_{int}:
  $\forall x\, \exists\, y.\ y < x \wedge P_{int}(y) \wedge \forall t.\ y < t < x \Rightarrow \neg P_{int}(t).$

The set of all integers is a model for P_{int}, therefore the above axiomatization is consistent. The set of elements satisfying P_{int} is necessarily infinite and does not admit a maximal or a minimal element. This is a direct consequence of the successor and predecessor axioms. More interestingly, this set is also necessarily countable. Indeed, since each point is isolated, there exists an application that maps the elements satisfying P_{int} to disjoint open intervals. Any set of disjoint

intervals in $\mathbb{R}$ with non-zero length is necessarily countable [18], since each of them contains a rational value that does not belong to the others.
It is now possible to define a successor relation on the real numbers satisfying P_{int} with the formula $Succ(x, y) = P_{int}(x) \wedge P_{int}(y) \wedge y < x \wedge \forall z.\, y < z < x \Rightarrow \neg P_{int}(z)$, i.e., x is the successor of y, or equivalently, y is the predecessor of x.
The axiomatization of P_{int} is, in fact, precise enough to have the following lemma.

Lemma 1. *For any model M of P_{int}, the set $M[P_{int}]$ is $<$-isomorphic to $\mathbb{Z}$.*

For convenience in the proof, we define 0_{int} as an arbitrary existentially quantified value that belongs to the set described by P_{int}.

Proof. Given a model M of the axiomatization of P_{int}, we need to define a bijection between the set $M[P_{int}]$ and $\mathbb{Z}$ that preserves order.

Let us define an application f from $M[P_{int}]$ to $\mathbb{Z}$. We set $f(0_{int}) = 0$, and then define recursively:

- $f(y) = f(x) + 1$ for each $x, y \in M[P_{int}]$ such that $y > 0_{int}$ and $Succ(y, x)$,
- $f(y) = f(x) - 1$ for each $x, y \in M[P_{int}]$ such that $y < 0_{int}$ and $Succ(x, y)$.

Thanks to the fact that every element of $M[P_{int}]$ has a unique predecessor and successor, it follows that f ranges over the whole set $\mathbb{Z}$, proving that f is surjective. Since it is clear that f preserves order, it follows that f is strictly increasing, and therefore injective. It remains to show that f is well defined for every element in $M[P_{int}]$.

If there exists some element $y \in M[P_{int}]$ for which f is not defined, it means that f is not well-defined, in the sense that there exists either an element $y > 0_{int}$ such that the interval $[0_{int}, y]$ contains an infinite number of elements satisfying P_{int}, or there exists an element $y < 0_{int}$ such that the interval $[y, 0_{int}]$ contains an infinite number of elements satisfying P_{int}. Since both cases are symmetric, we only address the former. There must exist a strictly increasing infinite series of elements in $M[P_{int}]$ bounded by y. Let us consider its limit $z \in \mathbb{R}$. Because there must exist an element of $M[P_{int}]$ smaller than z and arbitrarily close to z, it follows that z cannot have a predecessor, which contradicts an axiom. Therefore f is well-defined, and every element of $M[P_{int}]$ is associated to an integer number. The application f is therefore a bijection. □

3.2 Translating Formulas

We are now able to describe the satisfiability-preserving translation of formulas from UF1·IDL·IRO to UF1·RO. Consider a UF1·IDL·IRO formula φ. Without loss of generality, we assume that P_{int} does not appear in φ. The translation of φ is defined as

$$AXIOMS_{int}(P_{int}) \wedge [\![\varphi]\!]$$

where $AXIOMS_{int}(P_{int})$ is the conjunction of the axioms of P_{int}, and $[\![\cdot]\!]$ is a translation operator. This translation operator $[\![\cdot]\!]$ distributes over all Boolean operators and quantifiers, and corresponds to the identity transformation for most considered atoms, except in the following cases:

- $[\![x \in \mathbb{Z}]\!] = P_{int}(x)$;
- $[\![x - y \bowtie c]\!] = \exists z_0, \dots z_c \,.\, (y = z_0) \wedge (x \bowtie z_c) \wedge \bigwedge_{0 \le i < c} Succ(z_{i+1}, z_i)$, for $c \in \mathbb{N}$ and $\bowtie \in \{<, \le, =, \ge, >\}$. We assume that $z_0, \dots z_c$ are fresh variables w.r.t. x and y.

Example: $[\![x - y \le 2]\!] = \exists z_0, z_1, z_2 .\, y = z_0 \wedge Succ(z_1, z_0) \wedge Succ(z_2, z_1) \wedge x \le z_2$. Notice that we only deal with the case $c \in \mathbb{N}$ since every atom of the form $x - y \bowtie c$ with $c \in \mathbb{Z} \setminus \mathbb{N}$ and $\bowtie \in \{<, \le, =, \ge, >\}$ can be rewritten as $y - x \bowtie' -c$ with the following correspondences: $(\bowtie, \bowtie') \in \{(=,=), (<,>), (>,<), (\ge,\le), (\le,\ge)\}$.

3.3 Establishing Equisatisfiability

Given a UF1·IDL·IRO formula φ, the translation that we have introduced generates a corresponding UF1·RO formula ψ. To establish that they are equisatisfiable, we need to prove that if φ admits a model, then ψ also admits one, and reciprocally.

Lemma 2. *Given a* UF*1*·IDL·IRO *formula φ, consider its translation into* UF*1*·RO *$\psi = AXIOMS_{int}(P_{int}) \wedge [\![\varphi]\!]$. The formulas φ and ψ are equisatisfiable.*

Proof. If φ is satisfiable, let M be one of its models. Then, since ψ shares the same free variables and predicates than φ with the only addition of P_{int}, we can directly construct a model M' of ψ that is similar to M for the shared variables and predicates, and that interprets P_{int} so that $P_{int}(x)$ holds whenever $x \in \mathbb{Z}$. This is always possible since the only constraints on P_{int} generated by the construction of ψ are the axioms stated above.

If ψ is satisfiable, then there exists a model M of ψ. Let us construct a model M' of φ. Let $0_{int} \in \mathbb{R}$ be an arbitrary element of $M[P_{int}]$. We define an automorphism g of $\mathbb{R}$, such that $g(0_{int}) = 0$, and recursively $g(y) = g(x) + 1$ for $x, y \in M[P_{int}]$, $y > 0_{int}$ and $Succ(y, x)$, and $g(y) = g(x) - 1$ for $x, y \in M[P_{int}]$, $y < 0_{int}$ and $Succ(x, y)$. The automorphism g maps each open interval between the k-th and $(k+1)$-th successors (resp. predecessors) of 0_{int} in $M[P_{int}]$, onto the open interval $(k, k+1)$ (resp. $(-(k+1), -k)$) while preserving order.

M' is defined by $M'[x] = g(M[x])$ for each free variable x of the formula φ, and $M'[P] = \{g(x) \,|\, x \in M[P]\}$ for each uninterpreted predicate P of φ. No unary predicate atom can be violated by M' by definition. Furthermore, no order constraint can be violated by M' either since g preserves order. Regarding the difference logic constraints, the intermediate variables z_i introduced in the translation are necessarily mapped to values in $M[P_{int}]$ since the $Succ$ relation enforces this property. Hence for each such variable, we have $g(M[z_i]) \in \mathbb{Z}$. Intuitively, this ensures that in M' the difference between the values taken by the integer variables is consistent with the difference logic constraints. It follows that M' is a model of φ. □

4 Undecidability of UF1·RDL

The result presented in the previous section establishes a lower bound for the decidability of our family of fragments. A natural follow-up problem is to establish a corresponding upper bound, i.e., to find an extension of this logic that yields undecidability. We show here that, when combined with uninterpreted unary predicates, as soon as difference logic constraints on reals are allowed, the logic becomes undecidable.

We actually show a stronger result which is that a single unary predicate symbol is enough to yield undecidability. More precisely, we establish the undecidability of the restriction of UF1·RDL where only one predicate symbol is allowed, by reducing the halting problem of a Turing machine to the satisfiability problem over this restriction of UF1·RDL.

Theorem 2. *Satisfiability is undecidable for* UF1·RDL *with a single predicate.*

Corollary 1. *Satisfiability is undecidable for* UF1·RDL*.*

The remaining of this section is dedicated to proving Theorem 2. We consider w.l.o.g. Turing machines defined over an alphabet with only two symbols and no explicit blank symbol [16]. This choice leads to a simpler proof.

4.1 Definitions

The proof is by reduction from the halting problem for a Turing machine with a single bi-infinite tape, starting from a blank tape (i.e., a tape filled with the symbol 0). Consider a Turing machine $\mathcal{M} = (Q, \Sigma, q_I, q_F, \Delta)$, where

- Q is a finite nonempty set of states,
- Σ is the alphabet $\{0, 1\}$,
- $q_I \in Q$ is the initial state,
- $q_F \in Q$ is the halting state,
- $\Delta \subseteq \{(Q \setminus \{q_F\}) \times \Sigma \times Q \times \Sigma \times \{L, R\}\}$ is the transition relation, assumed to be total over its first two components, i.e., for any pair $(q, \alpha) \in (Q \setminus \{q_F\}) \times \Sigma$, there exists a tuple $(q, \alpha, q', \alpha', \lambda) \in \Delta$.

A *configuration* C of such a Turing machine is a triplet containing the current state q, the content of the tape $t \in \{0, 1\}^{\mathbb{Z}}$ and the position of the head $h \in \mathbb{Z}$. Since the machine starts from a blank tape, the initial configuration is $C_0 = (q_I, 0^{\mathbb{Z}}, 0)$.

A *run* ρ of length $n \in \mathbb{N}$ (resp. $n = +\infty$) of such a Turing machine is a finite (resp. infinite) sequence of configurations $(C_i)_{i \in [0,n]}$ (resp. $(C_i)_{i \in \mathbb{N}}$), such that for any two consecutive configurations $C_i = (q_i, t_i, h_i)$ and $C_{i+1} = (q_{i+1}, t_{i+1}, h_{i+1})$ there exists a transition $(q, \alpha, q', \alpha', \lambda) \in \Delta$ such that:

- $q = q_i$ and $q' = q_{i+1}$,
- $t_i[h_i] = \alpha$, i.e., the tape cell at position h_i contains the symbol α,
- $t_{i+1}[h_i] = \alpha'$,
- $t_{i+1}[k] = t_i[k]$, for every $k \in \mathbb{Z}$, $k \neq h_i$,
- $h_{i+1} = h_i + 1$ if $\lambda = R$, and $h_{i+1} = h_i - 1$ if $\lambda = L$.

A *halting run* is a finite run such that the state of its last configuration is the halting state q_F.

4.2 Encoding Runs

Our goal is to encode a run of a Turing machine (as described before), i.e., encode the state, the tape content, and the position of the head for each configuration of such a run. Starting from the initial configuration, we must also ensure the coherence of the run w.r.t. the Turing machine transition relation, by connecting every two consecutive configurations. Our idea is to define an infinite sequence of intervals on the real line, such that each interval contains the encoding of its corresponding configuration (i.e., the first interval will contain the first configuration of the run, and so on). Difference constraints can then be used to connect consecutive configurations.

Let $N = \lceil \log_2(|Q|) \rceil$. Each state $q \in Q$ of $\mathcal{M}$ can therefore be uniquely encoded with N Boolean values $b_1^q, \dots b_N^q$. We want to encode consecutive configurations of the Turing machine using a single predicate P over $\mathbb{R}$. In order to do so, we first need to describe a subset of $\mathbb{R}$ that will act as a grid supporting the encoding of the state, the tape content, and the head position of the current configuration.

We use the concept of linear ordering [15] to describe the shape of the grid. A *linear ordering* J is a totally ordered set, i.e., a set equipped with a binary relation $<$ which is irreflexive (for all j in J, $j \not< j$), asymmetric (for all j, k in J, if $j < k$, then $k \not< j$), transitive (for all i, j, k in J, if $i < j$ and $j < k$, then $i < k$), and complete (for all $j, k \in J$, either $j = k$, $j < k$, or $k < j$). The *order type* of a linear ordering J is the class of all linear orderings $<$-isomorphic to J. The order types of a singleton, the set composed of the N first natural numbers, $\mathbb{N}$, and $\mathbb{Z}$ are respectively denoted by 1, N, ω, and ζ. The concatenation of two linear orderings J and K (where their associated order relations are respectively $<_J$ and $<_K$) is denoted by $J+K$. It corresponds to the linear ordering composed of the set of pairs $\{(j,1) \mid j \in J\} \cup \{(k,2) \mid k \in K\}$, and equipped with the order relation $<$, defined by $(j_1,1) < (j_2,1)$ if $j_1 <_J j_2$, $(k_1,2) < (k_2,2)$ if $k_1 <_K k_2$, and $(j,1) < (k,2)$ for every $j \in J$ and $k \in K$. More generally, given two linear orderings J and K, the linear ordering $(J)^K$ is the set of pairs (j,k) with $j \in J$ and $k \in K$, with the order relation $<$ such that $(j_1,k_1) < (j_2,k_2)$ if either $k_1 <_K k_2$, or $k_1 =_K k_2$ and $j_1 <_J j_2$. These operators are naturally extended on order types. For instance, the order type $(\omega)^\omega$ is the class of all linear orderings $<$-isomorphic to $\mathbb{N}^2$.

The grid we consider is a linear ordering that is a subset of $\mathbb{R}$, of order type $\big(N + \zeta + 1 + \zeta\big)^\omega$. An ordering of order type $N + \zeta + 1 + \zeta$ within the interval $[0, 3)$ is depicted in Fig. 1. Each dot corresponds to a natural number and each vertical line corresponds to an element of the linear ordering. The first N points will support the encoding of a state. The first subordering that is $<$-isomorphic to $\mathbb{Z}$ (i.e., of order type ζ) will be used to encode the position of the head, while the second one will support the encoding of the tape content. The whole grid is composed of an infinite repetition of the subordering $N + \zeta + 1 + \zeta$ (i.e., it is repeated on the intervals $[3k, 3k+3)$ for all $k \in \mathbb{N}$), hence the ω exponent.

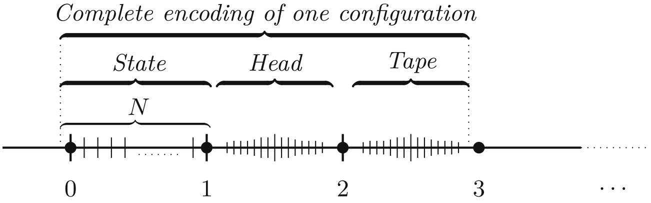


Fig. 1. A visual representation of a linear ordering of order type $N + \zeta + 1 + \zeta$.

4.3 Defining the Support of the Encoding

Let us first define concretely the support of the encoding of the Turing machine configurations. The difficulty lies in describing the grid using a single predicate P, without meddling with the actual encoding of the configurations afterwards. Our solution is to characterize the points that belong to the grid by enforcing that such a point is surrounded by an open interval where P is uniformly *true* on the left, and by an open interval where P is uniformly *false* on the right, such as depicted in Fig. 2. We do not specify yet how P behaves on x, as this is how the configurations will actually be encoded later.

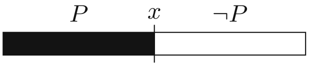


Fig. 2. The real number x belongs to the grid, since it is surrounded by a *true* (black) open interval on the left, and a *false* (white) open interval on the right.

Such a characterization is easy to express in our restriction of UF1·RDL:

$$\mathit{Support}(x) = (\exists y.\, y < x \wedge \forall z.\, y < z < x \Rightarrow P(z)) \wedge (\exists y.\, x < y \wedge \forall z.\, x < z < y \Rightarrow \neg P(z))$$

Let us now partially axiomatize the predicate P such that the set of *supporting* points constitutes a linear ordering of order type $\left(N + \zeta + 1 + \zeta\right)^{\omega}$:

(a) Let $\mathbf{0}$ be a variable and $\mathbf{1}, \mathbf{2}$ and $\mathbf{3}$ be respectively the +1-successor of $\mathbf{0}$, $\mathbf{1}$ and $\mathbf{2}$:
$\mathit{Axiom}_1 = (\mathbf{1} = \mathbf{0} + 1) \wedge (\mathbf{2} = \mathbf{1} + 1) \wedge (\mathbf{3} = \mathbf{2} + 1)$
These free variables are implicitly existentially quantified in the final formula.
Notice that the variable $\mathbf{0}$ can be interpreted as any real value, which only acts as a landmark for the beginning of the grid.

(b) $\mathbf{0}$, $\mathbf{1}$ and $\mathbf{2}$ are *supporting* points:
$\mathit{Axiom}_2 = \mathit{Support}(\mathbf{0}) \wedge \mathit{Support}(\mathbf{1}) \wedge \mathit{Support}(\mathbf{2})$

(c) P is uniformly true before $\mathbf{0}$, i.e., there are no supporting points before $\mathbf{0}$:
$\mathit{Axiom}_3 = \forall x.\, x < \mathbf{0} \Rightarrow P(x)$

(d) There are exactly $N-2$ *supporting* points within the interval $(\mathbf{0},\mathbf{1})$:
$\mathit{Axiom}_4 = \exists x_1, x_2, \dots x_N.\, x_1 = \mathbf{0} \wedge x_N = \mathbf{1}$
$\qquad \wedge \bigwedge_{1 \le i < N} \left(\mathbf{0} \le x_i < \mathbf{1} \wedge \mathit{Succ}_{\mathit{Supp}}(x_{i+1}, x_i)\right)$
where $\mathit{Succ}_{\mathit{Supp}}(x,y)$ is a formula that states that x is the first *supporting* real value that is strictly greater than y, i.e., x is the successor of y on the grid. It is defined as follows:
$\mathit{Succ}_{\mathit{Supp}}(x,y) = y < x \wedge \mathit{Support}(x) \wedge \mathit{Support}(y) \wedge \forall z.\, y < z < x \Rightarrow \neg\, \mathit{Support}(z)$
We also define an analogous formula to express that x is the predecessor of y: $\mathit{Pred}_{\mathit{Supp}}(x,y) = \mathit{Succ}_{\mathit{Supp}}(y,x)$.

(e) The set of *supporting* points within $(\mathbf{1},\mathbf{2})$ is $<$-isomorphic to $\mathbb{Z}$. This is done similarly to the axiomatization of P_{int} (cf. Section 3.1). But because $\mathbf{1}$ (resp. $\mathbf{2}$) is a *supporting* point, there must exist a uniformly false (resp. true) interval of P at its right (resp. left) where no other *supporting* points can appear. All the *supporting* points will therefore be constrained to appear within a smaller interval (b_1, b_2) with $\mathbf{1} < b_1 < b_2 < \mathbf{2}$, as illustrated in Fig. 3.

$$\begin{aligned}
\mathit{Axiom}_5 = {} & [\exists b_1, b_2.\, \mathbf{1} < b_1 < b_2 < \mathbf{2}] && (1)\\
& \wedge\ [\forall x.\, (b_1 < x < b_2) \Rightarrow \exists y.\, x < y < b_2 \wedge \mathit{Support}(y) \\
& \qquad\qquad \wedge \forall z.\, x < z < y \Rightarrow \neg \mathit{Support}(z)] && (2)\\
& \wedge\ [\forall x.\, (b_1 < x < b_2) \Rightarrow \exists y.\, b_1 < y < x \wedge \mathit{Support}(y) \\
& \qquad\qquad \wedge \forall z.\, y < z < x \Rightarrow \neg \mathit{Support}(z)] && (3)\\
& \quad [\forall x.\, (1 < x < 2 \wedge \mathit{Support}(x)) \Rightarrow b_1 < x < b_2] && (4)
\end{aligned}$$

This axiom can be broken down into these elementary pieces:
(1) there exists an open interval (b_1, b_2) such that $\mathbf{1} < b_1 < b_2 < \mathbf{2}$,
(2) each real value in (b_1, b_2) has a *supporting* successor,
(3) each real value in (b_1, b_2) has a *supporting* predecessor,
(4) there are no *supporting* points within $(\mathbf{1}, b_1)$, nor within $(b_2, \mathbf{2})$.

(f) The pattern of *supporting* points within $(\mathbf{1},\mathbf{2})$ is repeated onto the interval $(\mathbf{2},\mathbf{3})$ with an exact offset of 1:
$\mathit{Axiom}_6 = \forall x.\, \mathbf{1} < x < \mathbf{2} \Rightarrow (\mathit{Support}(x) \Leftrightarrow \mathit{Support}(x+1))$

(g) The pattern of *supporting* points within $[\mathbf{0},\mathbf{3})$ is repeated onto every interval $[\mathbf{3k}, \mathbf{3k}+\mathbf{3})$ for $k \in \mathbb{N}$:
$\mathit{Axiom}_7 = \forall x.\, x \ge \mathbf{0} \Rightarrow (\mathit{Support}(x) \Leftrightarrow \mathit{Support}(x+3))$

Notice that for Axiom_7, it is not enough that a similar pattern appears within each interval $[\mathbf{3k}, \mathbf{3k}+\mathbf{3})$: there must be an exact offset of 3 with the previous interval. This is mandatory to connect two consecutive configurations and ensure that they are coherent with the transition relation of the Turing machine, as defined later. The same goes for Axiom_6, where the exact offset of 1 will allow to connect the position of the head to the tape content within a single configuration.

The formula $AXIOMS_{Supp} = \bigwedge_{1 \leq k \leq 7} Axiom_k$ axiomatizes the predicate P.

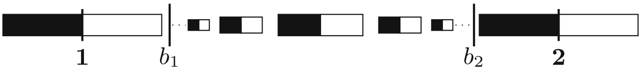


Fig. 3. The points of the grid surrounded by open *true* (black) and *false* (white) intervals within $(\mathbf{1}, \mathbf{2})$.

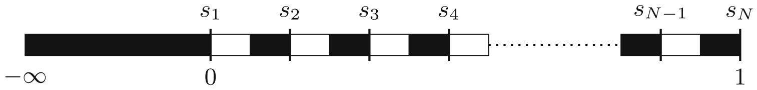


Fig. 4. A model for the axiomatization of P over the interval $(-\infty, 1)$.

Lemma 3. *The formula $AXIOMS_{Supp}$ is consistent.*

The proof sketch below provides the key ideas to construct a model of $AXIOMS_{Supp}$. The complete construction is described in [2].

Proof. Let us construct a subset S of $\mathbb{R}$ that is a model of $AXIOMS_{Supp}$. Firstly, we make every negative number belong to S, which ensures that there do not exist negative supporting points. The interval $[0, 1]$ is then cut into $2N - 2$ intervals of equal length, which alternate between being included in S, and being disjoint from S. This ensures the existence of exactly $N - 1$ supporting points within the interval $(-\infty, 1)$, 0 being the first; 1 will be considered later. These $N - 1$ supporting points are referred to as $s_1, s_2, \ldots s_{N-1}$ and are depicted in Fig. 4. Recall that the supporting points are exactly those surrounded by an interval of S (i.e., black on the figure) on the left, and an interval disjoint from S (i.e., white) on the right.

In order to make the real value 1 the N-th supporting point, it is enough to make an interval on its right disjoint from S, e.g., the interval $(1, 1 + \frac{1}{4})$. Symmetrically, we make the interval $(2 - \frac{1}{4}, 2)$ included in S, satisfying the left part of the requirement for the real value 2 to be a supporting point.

We further characterize S such that the set of supporting points within the interval $(1 + \frac{1}{4}, 2 - \frac{1}{4})$ is $<$-isomorphic to $\mathbb{Z}$. This can be done by partitioning the

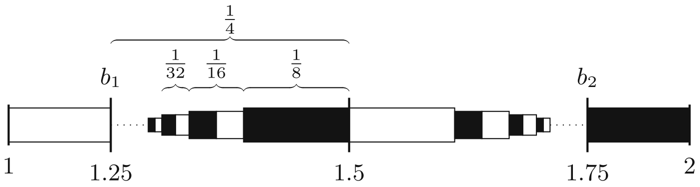


Fig. 5. A model for the axiomatization of P over the interval $(1, 2)$.

open interval $(1+\frac{1}{4}, 2-\frac{1}{4})$ into a bi-infinite sequence of open intervals alternating between being included and disjoint from S, as depicted in Fig. 5.

The whole pattern described on the interval $(1, 2)$ can be directly transposed onto the interval $(2, 3)$ with an exact offset of $+1$. Similarly, the distribution of S over the interval $(0, 3)$ can be transposed onto every interval $(3k, 3k+3)$ with an offset of $+3k$, for $k > 0$. The only real values for which we do not describe their relation with S are the points surrounded by an interval included in S on one side, and an interval disjoint from S on the other side. These points never conflict with the axiomatization $\mathit{AXIOMS}_{\mathit{Supp}}$ which only deals with non-empty open intervals.

By construction, S satisfies each axiom of the formula $\mathit{AXIOMS}_{\mathit{Supp}}$, and is therefore a model of this formula. □

4.4 Encoding a Configuration of the Turing Machine

Now that the supporting grid has been properly defined, the actual encoding of a given configuration can be addressed. That is, the state, the tape content and the head position of the $(k+1)$-th configuration of a run are encoded on the supporting points contained within the interval $[3k, 3k+3)$.

Encoding the State. Encoding the state of a given configuration is rather direct since we defined the grid to contain N consecutive supporting points within every interval $[3k, 3k+1]$ for $k \in \mathbb{N}$, that can support the encoding of a state. We only need to indicate that we start reading the encoding on a multiple of 3. However the logic UF1·RDL does not allow to express periodicity constraints on variables. Nevertheless, thanks to our axiomatization, 0 and every other positive multiple of 3 are the only points that simultaneously have no supporting predecessor, while admitting a supporting successor. These properties are expressible as follows:
$\mathit{NoPred}_{\mathit{Supp}}(x) = \forall z.\, (z < x \wedge \mathit{Support}(z)) \Rightarrow \exists y.\, z < y < x \wedge \mathit{Support}(y)$
$\mathit{HasSucc}_{\mathit{Supp}}(x) = \exists z.\, x < z \wedge \mathit{Support}(z) \wedge \forall y.\, x < y < z \Rightarrow \neg \mathit{Support}(y)$
For convenience, we introduce the formula *EncodingBegins* to characterize a real value x on which the encoding of a state starts:
$\mathit{EncodingBegins}(x) = \mathit{Support}(x) \wedge \mathit{NoPred}_{\mathit{Supp}}(x) \wedge \mathit{HasSucc}_{\mathit{Supp}}(x)$
Furthermore, the formula State_q expresses that a state $q \in Q$ is encoded on a given real number x and its $N-1$ supporting successors:

$$\begin{aligned}\mathit{State}_q(x) = {} & \mathit{EncodingBegins}(x) \wedge \exists y_1, \dots y_N.\, x = y_1 \\ & \wedge \bigwedge_{1 \le i < N} \mathit{Succ}_{\mathit{Sup}}(y_{i+1}, y_i) \wedge \bigwedge_{1 \le i \le N} P(y_i) = b_i^q\end{aligned}$$

where $P(y_i) = b_i^q$ is a shorthand for $P(y_i)$ if $b_i^q = \top$, and $\neg P(y_i)$ if $b_i^q = \bot$.

Encoding the Head Position. The position of the head is encoded in the second part of the grid, that is, in the interval $(3k+1, 3k+2)$ for the $(k+1)$-th configuration (cf. Fig. 1). The grid on this interval is <-isomorphic to $\mathbb{Z}$. Each element of this subordering will correspond to a position of the tape. When the predicate P is true at such a point, it means that the head points towards that cell. Since the Turing machines that we consider here have a single read/write head, it must point towards a unique cell for each configuration. Therefore P must be true only for a single element of that subordering.

Encoding the Tape Content. Similarly, the tape content is encoded in the third part of the grid, that is, in the interval $(3k+2, 3k+3)$ for the $(k+1)$-th configuration (cf. Fig. 1). Again, the grid on this interval is <-isomorphic to $\mathbb{Z}$. And again, each element x of this subordering will correspond to a cell of the tape, matching the cell that corresponds to $x-1$ in the head position interval. Figure 6 illustrates the connections between the suborderings, within a single configuration and with the next one. The idea of the encoding is to simply set the value of P to true on the elements of the subordering that correspond to cells containing a 1, and to false for cells containing a 0.

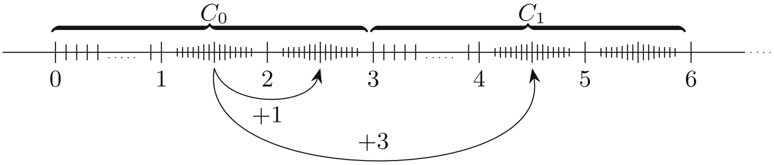


Fig. 6. The first two consecutive configuration encodings.

4.5 Enforcing a Valid Run

Let us now define formally the formulas characterizing an accepting run of $\mathcal{M}$. We will decompose the global formula into three main parts: the initial conditions $\mathit{START}_{\mathcal{M}}$, the conditions on the transitions $\mathit{STEP}_{\mathcal{M}}$ and the halting condition $\mathit{END}_{\mathcal{M}}$. For the sake of clarity, we use capital letters for these higher-level formulas.

The initial conditions of $\mathcal{M}$ are that the state encoded on $\mathbf{0}$ and its $N-1$ supporting successors is the initial state q_0, that the head points towards a unique initial unspecified cell of the tape, and finally that the tape is initially filled with 0's. These conditions are expressed by the following formula:

$$\begin{aligned}\mathit{START}_{\mathcal{M}} = \mathit{State}_{q_0}(\mathbf{0}) \wedge \big[&\exists y.\, \mathbf{1} < y < \mathbf{2} \wedge \mathit{Support}(y) \wedge P(y)\\ &\wedge \forall x.\, (\mathbf{1} < x < \mathbf{2} \wedge \mathit{Support}(x) \wedge P(x)) \Rightarrow x = y\big]\\ \wedge\ &\big[\forall y.\, (\mathbf{2} < y < \mathbf{3} \wedge \mathit{Support}(y)) \Rightarrow \neg P(y)\big]\end{aligned}$$

The requirements on the transition are more complex. Intuitively, if before reaching the step $i \in \mathbb{N}$, we have not yet encountered the halting state q_F, then we must ensure that the configuration at Step i can be obtained from the configuration at the previous step $i-1$ by following a transition $(q, \alpha, q', \alpha', \lambda) \in \Delta$. The overall formula for this condition is the following:

$$\begin{aligned} \mathit{STEP}_{\mathcal{M}} = \forall y.\, (y > 0 \wedge \mathit{EncodingBegins}(y) \wedge \mathit{NotEnded}_{\mathcal{M}}(y)) \\ \Rightarrow \exists x.\, y = x + 3 \wedge \mathit{Transition}_{\mathcal{M}}(x, y) \end{aligned}$$

The subformula $\mathit{NotEnded}_{\mathcal{M}}(y)$ expresses that no valid real value prior to y (i.e., a positive multiple of 3 strictly smaller than y) encodes the halting state. This formula is defined by:

$$NotEnded_{\mathcal{M}}(y) = \forall x.\, (x < y \wedge \mathit{EncodingBegins}(x)) \Rightarrow \neg(\mathit{State}_{q_F}(x))$$

The subformula $\mathit{Transition}_{\mathcal{M}}(x, y)$ expresses that there exists a transition $(q, \alpha, q', \alpha', \lambda) \in \Delta$ that allows to move in one step from the configuration encoded at x (i.e., that the encoding of the configuration starts exactly on x), to the configuration corresponding to y. To improve readability, we decompose the condition on the transition relation as follows:

$$\begin{aligned} &\mathit{Transition}_{\mathcal{M}}(x, y) = \\ &\bigvee_{(q,\alpha,q',\alpha',\lambda)\in\Delta} \Big[\mathit{State}_q(x) \wedge \mathit{State}_{q'}(y) \wedge \mathit{Tape}_{\alpha,\alpha'}(x, y) \wedge \mathit{Head}_{\lambda}(x, y)\Big] \end{aligned}$$

For a given transition $(q, \alpha, q', \alpha', \lambda) \in \Delta$, the conditions on the states, tape and head are expressed as follows:

- The state q must be encoded on the real value x, and the state q' on y: $\mathit{State}_q(x) \wedge \mathit{State}_{q'}(y)$
- The tape must contain $\alpha \in \{0, 1\}$ at the position of the head for the step corresponding to x. Additionally, for the step corresponding to y, the tape must contain α' at the previous position of the head, and remain unchanged at all other positions.

  $$\begin{aligned} \mathit{Tape}_{\alpha,\alpha'}(x, y) = \big[&\forall z.\, (x + 1 < z < x + 2 \wedge \mathit{Support}(z) \wedge P(z)) \\ &\Rightarrow P(z + 1) = \alpha \wedge P(z + 4) = \alpha'\big] \\ \wedge\big[&\forall z.\, (x + 1 < z < x + 2 \wedge \mathit{Support}(z) \wedge \neg P(z)) \Rightarrow (P(z + 1) \Leftrightarrow P(z + 4))\big] \end{aligned}$$

  where $P(z + k) = \alpha$ is a shorthand for $\exists u.\, u = z + k \wedge P(u)$ if $\alpha = 1$, and $\exists u.\, u = z + k \wedge \neg P(u)$ if $\alpha = 0$. The "$+1$" operator allows us to connect the encoding of the head position with the encoding of the tape content within the same configuration. The "$+4$" operator does the same while jumping to the next configuration (cf. Fig. 4). Notice that this formula does not involve y; it assumes (rightfully, given the formula $\mathit{STEP}_{\mathcal{M}}$) that the equality $y = x+3$ holds.
- The head is moved in the direction specified by $\lambda \in \{L, R\}$, i.e., left for L and right for R. This can be expressed by exploiting the predecessor and successor relations defined for supporting real values.

  $$\begin{aligned} \mathit{Head}_{\lambda}(x, y) = \forall z.\, (x + 1 < z < x + 2 \wedge \mathit{Support}(z) \wedge P(z)) \\ \Rightarrow \exists v.\, f_{\lambda}(v, z + 3) \wedge P(v) \wedge \neg P(z) \end{aligned}$$

where $f_R = Succ_{Supp}$ and $f_L = Pred_{Supp}$. Since in the initial configuration of the Turing machine the head points towards a single cell, the formula $Head_\lambda$ ensures that this remains the case throughout every run of the Turing machine.

Finally, the existence of a halting run is expressed by the formula:

$$END_{\mathcal{M}} = \exists x.\, State_{q_F}(x)$$

The global formula that expresses that the Turing machine $\mathcal{M}$ halts on some run encoded by the value of the predicate P is the following:

$$HALT_{\mathcal{M}} = START_{\mathcal{M}} \wedge STEP_{\mathcal{M}} \wedge END_{\mathcal{M}} \wedge AXIOMS_{Supp}$$

where $AXIOMS_{Supp}$ is the axiomatization of the supporting points as described in Sect. 4.3.

By construction, satisfiability of the global formula $HALT_{\mathcal{M}}$ is equivalent to the existence of a halting run for the Turing machine $\mathcal{M}$. It follows that the satisfiability problem for UF1·RDL is undecidable, which proves Theorem 2.

5 Conclusion

This work provides a lower and an upper bound for the decidability of first-order fragments with quantifiers mixing uninterpreted unary predicates and weak forms of real arithmetic. This draws a precise picture of the frontier of decidability in fragments mixing real arithmetic and uninterpreted predicates.

We proved the decidability of the fragment UF1·IDL·IRO, where uninterpreted unary predicates, order constraints between real and integer variables, and difference logic constraints between integer variables are allowed. This result is a consequence of the already established decidability of its restriction UF1·RO, where only uninterpreted unary predicates and order constraints between real values are allowed. To the best of our knowledge, there does not exist yet a practical decision procedure for UF1·RO.

There exist fragments of arithmetic that are more expressive than difference logic, but still weaker than full Presburger arithmetic. It would be interesting to investigate if decidability for these is preserved in presence of uninterpreted unary predicates. Note however that our proof of decidability strongly relies on the translation of the constraints into the first-order theory of order over $\mathbb{R}$, with unary predicates. This translation is not suitable for, e.g., constraints of the form $x + y \bowtie 0$, where x and y are variables, and $\bowtie \in \{<, \leq, =, \geq, >\}$.

In another result, we established the undecidability of the fragment UF1·RDL, where uninterpreted unary predicates and difference logic constraints between real variables are allowed. It is worth mentioning that this result can be adapted straightforwardly to the same logic interpreted over the domain $\mathbb{Q}$.

Our long term goal is to design an effective decision procedure for the decidable fragment. Complexity results have been established [6,13,14] for the temporal logic counterpart of the theory of order, to which we reduce the decidability of our fragment of interest. We are currently designing a decision procedure relying on the concept of automata on linear orderings introduced in [3]. We hope that the insight we obtained through this decision procedure will eventually guide the design of new powerful instantiation techniques for SMT in a more expressive context, and that these techniques will happen to be complete in particular for this decidable fragment.

Acknowledgments. We are thankful to Tanja Schindler and the reviewers of this paper and of our previous work-in-progress workshop paper for their comments.

References

1. Boigelot, B., Fontaine, P., Vergain, B.: Decidability of difference logics with unary predicates. In: Proceedings, 7th International Workshop on Satisfiability Checking and Symbolic Computation (2022)
2. Boigelot, B., Fontaine, P., Vergain, B.: Decidability of difference logic over the reals with uninterpreted unary predicates. arXiv preprint arXiv:2305.15059 (2023)
3. Bruyère, V., Carton, O.: Automata on linear orderings. J. Comput. Syst. Sci. **73**(1), 1–24 (2007)
4. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Logic, Methodology and Philosophy of Science (1962)
5. Burgess, J.P., Gurevich, Y.: The decision problem for linear temporal logic. Notre Dame J. Formal Logic **26**(2), 115–128 (1985)
6. Cristau, J.: Automata and temporal logic over arbitrary linear time. In: FSTTCS 2009. LIPIcs, vol. 4, pp. 133–144 (2009)
7. Downey, P.J.: Undecidability of Presburger arithmetic with a single monadic predicate letter. Center for Research in Computer Technology, Harvard University, Technical report (1972)
8. Enderton, H.B.: A Mathematical Introduction to Logic, 2nd edn. Academic Press, Boston (2001)
9. Gurevich, Y., Shelah, S.: Monadic theory of order and topology in ZFC. Ann. Math. Logic **23**(2–3), 179–198 (1982)
10. Haase, C.: A survival guide to Presburger arithmetic. ACM SIGLOG News **5**(3), 67–82 (2018)
11. Halpern, J.Y.: Presburger arithmetic with unary predicates is Π_1^1 complete. J. Symbolic Logic **56**(2), 637–642 (1991)
12. Matiyasevich, Y.V.: Hilbert's Tenth Problem. MIT Press, Cambridge (1993)
13. Rabinovich, A.: Temporal logics over linear time domains are in PSPACE. Inf. Comput. **210**, 40–67 (2012)
14. Reynolds, M.: The complexity of temporal logic over the reals. Ann. Pure Appl. Logic **161**(8), 1063–1096 (2010)
15. Rosenstein, J.G.: Linear Orderings. Academic Press, Cambridge (1982)
16. Shannon, C.E.: A universal Turing machine with two internal states. Automata Stud. **34**, 157–165 (1956)
17. Shelah, S.: The monadic theory of order. Ann. Math. **102**(3), 379–419 (1975)
18. Sierpiński, W.: Cardinal and ordinal numbers, 2nd edn. PWN, Warszawa (1965)
19. Speranski, S.O.: A note on definability in fragments of arithmetic with free unary predicates. Arch. Math. Log. **52**(5–6), 507–516 (2013)
20. Tarski, A.: A Decision Method for Elementary Algebra and Geometry, 2nd edn. University of California Press, Berkeley (1951)

Incremental Rewriting Modulo SMT

Gerald Whitters[1(✉)], Vivek Nigam[3], and Carolyn Talcott[2]

[1] UPENN, Philadelphia, USA
whitters@seas.upenn.edu
[2] SRI International, Menlo Park, USA
[3] Federal University of Paraíba, João Pessoa, Brazil

Abstract. Rewriting Modulo SMT combines two powerful automated deduction techniques (1) rewriting and (2) SMT-solving. Rewriting enables the specification of behavior of systems using rewriting rules, while SMT theories specify system properties. Rewriting Modulo SMT is enabled by combining existing tools, such as Maude and SMT solvers. Search algorithms used for carrying out Rewriting Modulo SMT, however, cannot exploit the incremental solving features available in SMT solvers as they are based on breadth-first search. This paper addresses this limitation by proposing Incremental Rewriting Modulo SMT Theories, which is a syntactical restriction to rewriting rules. This restriction turns out to naturally be used in several applications of Rewriting Modulo SMT, including the verification of algorithms, cyber-physical systems, and security protocols. Moreover, we propose a Hybrid-Search algorithm for Incremental Rewriting Modulo SMT Theories that combines breadth-first search and depth-first search, thus enabling incremental SMT-solving. We demonstrate through a collection of existing benchmarks that the Hybrid-Search algorithm can achieve a 10 times performance improvement in verification times.

1 Introduction

Rewriting modulo SMT [14] is the result of the combination of two powerful automated deduction methods: rewriting logic and SMT-solving. It is supported by the integration [11] of powerful tools, such as Maude [6] and Z3 [8]. During rewriting, a set of constraints on the symbols appearing in a term are generated. These constraints can be, for example, non-linear arithmetic constraints that specify possible values that can be assumed by the configuration parameters. Demonstrating properties of such specifications amounts to search using these rewrite rules and satisfiability checking of the accumulated constraints using SMT solvers. Rewriting modulo SMT has been successfully applied in several case-studies from several domains, including safety of cyber-physical systems (CPSes) [13]; verification of algorithms [2]; and for network security analysis [16].

One important aspect that has not been addressed until now is how to exploit an SMT solver's capability of incrementally solving problems. In this solving method, instead of checking for the satisfiability of a formula from scratch, it reuses data previously computed by prior checks. For example, if the satisfiability

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 560–576, 2023.
https://doi.org/10.1007/978-3-031-38499-8_32

of a formula b has been checked, the check on $\mathsf{b} \wedge \mathsf{b}_I$ may re-use the intermediate results obtained while checking for the satisfiability of b. It has been shown that incremental solving can greatly improve performance by a factor of 2–5 times [10].[1]

The search algorithms used to implement rewriting modulo SMT are similar to those implemented in the Maude search engine [6]. They use a breadth-first search (BFS) algorithm with memoization techniques in order to improve performance. This type of search seems incompatible with incremental solving as constraints appearing in different branches of the search tree are generated under different conditions. Thus, it is hard to define what the increment (b_I mentioned above) would be.

This paper's goal is to enable rewriting modulo SMT that can exploit incremental solving. To achieve this, we make the following contributions:

- **Incremental Rewriting Modulo SMT** by identifying a class of rewrite rules that are amenable to incremental solving. More specifically, rewrite rules are applied to terms containing symbols paired with a set of boolean terms constraining the values of these symbols. Moreover, any rewrite rule can only add new constraints, i.e., not change the existing set of constraints on the term that is being rewritten. We show that a variety of theories used in published case studies can be seen to be amenable to incremental solving.
- **A Hybrid Search Algorithm for Incremental Theories** which combines breadth and depth-first search (DFS) strategies. The combination is parameterized by a level of depth parameter which specifies how many depth-first search steps shall be performed before switching to a breath-first search. The proposed hybrid search algorithm enjoys the benefits of BFS, namely better coverage as it alternates through different branches of the search tree, and the benefits of DFS, namely incremental solving.

We carried out a collection of experiments (the case studies mentioned above) on algorithm verification, cyber-physical systems verification, and network security analysis. The experiments show that in all these benchmarks, the hybrid search algorithm outperforms current BFS techniques, in some experiments achieving a 10 factor performance improvement.

Section 2 illustrates the problems of existing BFS methods for Rewriting Modulo SMT and proposes Incremental Rewriting Theories which formalizes the notion of increments. Section 3 describes the Hybrid algorithm proposed illustrating how it enables incremental SMT solving. Section 4 describes experiments that compare different search mechanisms (BFS, DFS, and Hybrid) on existing benchmarks from the literature. Finally, we conclude by discussing Related Work in Sect. 5 and Future Work in Sect. 6.

2 Incremental Rewriting Modulo SMT

Rewriting logic [12] is a logical formalism that is based on two ideas: states of a system are represented as elements of an algebraic data type, specified in an

[1] Albeit, incremental solving can also reduce performance depending on the theories that are used.

equational theory, and the behavior of a system is given by local transitions between states described by rewrite rules. A rewrite rule has the form $\mathsf{t} \to \mathsf{t}'$ if b, where t and t' are terms possibly containing variables and b is a condition (a boolean term). Such a rule applies to a system in state s (a ground term) if t can be matched to a part of s by supplying the right values for the variables, and if the condition b holds when supplied with those values. In this case, the rule can be applied by replacing the part of s matching t by t' using the matching values for the variables in t'.

Maude is a language and tool based on rewriting logic [6]. Maude provides a high performance rewriting engine featuring matching modulo associativity, commutativity, and identity axioms; and search and model-checking capabilities. Thus, given a specification S of a concurrent system, one can execute S to find one possible behavior; use search to see if a state meeting a given condition can be reached; or model-check S to see if a temporal property is satisfied, and if not, to see a computation that is a counterexample.

Symbolic rewriting modulo SMT [13,14] allows rewriting symbolic states (t, b), where t is a term possibly containing variables and b a boolean term constraining the allowed values of variables of t. The symbolic state (t, b) represents the set of (concrete) states that are instances of t such that the instantiating substitution satisfies b. Thus a rewrite to a symbolic state $\mathsf{t}', \mathsf{b}')$ such that b' is not satisfiable represents the empty set of concrete rewrites and satisfiability can be checked at each step to avoid useless work. This independent of checking that a goal is satisfied by a symbolic state. To implement symbolic rewriting in Maude, variables are replaced by symbols, treated as constants by Maude, and translated as variables when using an SMT solver to check satisfiability of the constraint. Symbolic rewriting allows us to reason about open systems, and to reason about all (possibly infinitely many) instances of a configuration.

Verification problems are expressed as reachability problems expressed as statements for the form

$$\mathsf{search}(\mathsf{t}_0, \mathsf{b}_0) \Rightarrow (\mathsf{t}', \mathsf{b}') \ \mathsf{such\ that\ goalCond}(\mathsf{t}', \mathsf{b}')$$

where $(\mathsf{t}', \mathsf{b}')$ is a pattern and $\mathsf{goalCond}$ is a boolean function that checks whether a state satisfies some condition. Typically, $\mathsf{goalCond}(\mathsf{t}', \mathsf{b}')$ also makes calls to the SMT solver to check whether some constraints derived from b' are satisfiable.

As illustrated by Fig. 1, Rewriting Modulo SMT implementations [11] traverse the search tree derived from the rewrite rules using BFS-based algorithms. At each step, e.g., $(\mathsf{t}_0, \mathsf{b}_0) \to (\mathsf{t}_1, \mathsf{b}_1)$, the engine checks for the satisfiability of the condition b_1. If the check fails, then search backtracks following BFS strategy. Otherwise, if the check succeeds, then the engine checks (1) whether $(\mathsf{t}_1, \mathsf{b}_1)$ matches the pattern $(\mathsf{t}', \mathsf{b}')$ and (2) if this is the case, it checks the condition $\mathsf{goalCond}(\mathsf{t}_1, \mathsf{b}_1)$, which may make further calls to the SMT solver, written as $\mathsf{SMT}(\mathsf{goalCond}(\mathsf{t}_1, \mathsf{b}_1))$. If $\mathsf{goalCond}$ returns true, then a solution for the reachability problem is found. Otherwise, the algorithm continues search following BFS.

From the sequence of calls to the SMT solver, one can observe the following difficulties of exploiting incremental SMT solving when using BFS based search strategy:

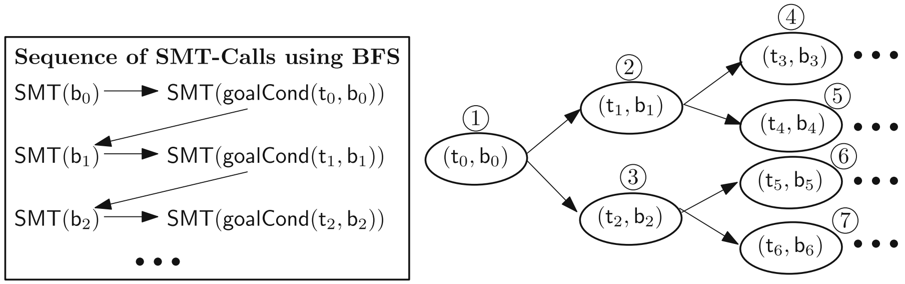


Fig. 1. Illustration of the search tree and SMT-calls when using Rewriting Modulo SMT following a BFS algorithm. The sequence of SMT-calls of a BFS algorithm is depicted to the left, where $\mathsf{SMT}(\mathsf{goalCond}(\mathsf{t}_i, \mathsf{b}_i))$ denotes possible SMT-calls required by the goal condition $\mathsf{goalCond}$. The numbers inside the circles specify the order in which nodes are traversed.

- **Definitions of Increments:** Given the generality of the accepted theory, it is not possible for the search engine to determine whether constraints, b_1 and b_2, used in subsequent calls to the SMT, $\mathsf{SMT}(b_1)$ and $\mathsf{SMT}(b_2)$, are constructed using some increment, i.e., whether $\mathsf{b}_2 = \mathsf{b}_1 \wedge \mathsf{b}_{1,2}$. This is because b_1 and b_2 are derived by applying different instances of rules which normally add/modify constraints in different ways.
- **Not possible to chain incremental calls:** As it is not possible to define increments when using rewrites rules in general, it is not possible to effectively use incremental solving by chaining calls, such as in $\mathsf{SMT}(\mathsf{b}_1); \mathsf{SMT}(\mathsf{b}_1 \wedge \mathsf{b}_{1,2}); \mathsf{SMT}(\mathsf{b}_1 \wedge \mathsf{b}_{1,2} \wedge \mathsf{b}_{2,3}) \ldots$.

To address this problem, we introduce a special class of rewrite theories, called *Incremental Rewrite Theories*.

Definition 1. *An incremental rewrite theory is a rewrite theory specification $\langle \Sigma, \mathcal{E}, \mathcal{R} \rangle$ where Σ is a typed alphabet; $\mathcal{E}$ is an equational theory; and $\mathcal{R}$ is a set of rewrite rules of the forms:*

$$(\mathsf{t}, \mathsf{b}) \to (\mathsf{t}_1, \mathsf{b} \wedge \mathsf{b}_I) \qquad \textit{and} \qquad (\mathsf{t}, \mathsf{b}) \to (\mathsf{t}_1, \mathsf{b} \wedge \mathsf{b}_I) \ \textsf{\textit{if cond}}$$

where t, t_I are well-formed terms; b, b_I are boolean formulas (in a given theory); and cond is a conjunction of equations. [2]

[2] The rule on the left is an unconditional rewrite rule that can be applied whenever it matches a subterm of the current state. The rule on the right is conditional. cond specifies conditions under which the rule can be applied. The condition is checked using using the equational theory to determine if the equations are satisfied by a candidate matching substitution. The term (t, b) represents a set of values, namely all instances for which the constraint b is true. A constraint solver is used to determine if b is satisfiable, that is, if the set of values is non-empty. In brief, the difference between b and cond is how they are used in reasoning.

The verification problem for incremental problems is a specialized reachability problem as defined below.

Definition 2. *Let $\mathcal{T}$ be an incremental rewrite theory. An incremental reachability problem over $\mathcal{T}$ is of the form:*

$$\mathit{search}(\mathsf{t}_0, \mathsf{b}_0) \Rightarrow (\mathsf{t}', \mathsf{b}') \text{ such that } \mathit{goalTerm}(\mathsf{t}') \text{ and } \mathsf{SMT}(\mathsf{b}' \wedge \mathsf{b}_I)$$

where $\mathit{goalTerm}$ is a function that takes a term and returns a boolean value and $\mathsf{b}_I = \mathsf{goal}(\mathsf{t}')$ is a boolean formula constructed from t'.

The following three examples illustrate how incremental theories can model different types of systems. These examples are based on specifications from the literature [2,13,16]. For ease of exposition, we simplify the rules in the description below. In Sect. 4, the full specifications from the literature are used in our experiments.

Example 1. This example is based on the work [2] for verification of the CASH scheduling algorithm [4]. In this algorithm, each task has a worst-case execution time. Whenever a task is completed before its deadline, the unused processing time is added to a global queue of unused budget, which can then be used by other tasks. Rewriting modulo SMT has been used to verify whether it is possible for a task to miss its deadline [2]. In particular, constraints keep track of the processing times and the available time budgets.

It turns out that the specification of this algorithm as rewrite rules and the verification problem are an incremental rewrite theory and an incremental reachability problem, respectively. For example, the following rule specifies when a deadline is missed:

$$\begin{aligned}
&(\langle \mathsf{id}_1 : \mathsf{global} \mid \mathsf{deadlineMiss} : \mathit{missStat}, \mathsf{Ats}\rangle,\\
&\langle \mathsf{id}_0 : \mathsf{server} \mid \mathsf{state} : st, \mathsf{usedBudget} : t, \mathsf{timeDeadline} : t_1, \mathsf{maxBudget} : n\rangle \ \mathsf{rest}, \mathsf{b})\\
&\to (\langle \mathsf{id}_1 : \mathsf{global} \mid \mathsf{deadlineMiss} : \mathsf{true}, \mathsf{Ats}\rangle,\\
&\langle \mathsf{id}_0 : \mathsf{server} \mid \mathsf{state} : st, \mathsf{usedBudget} : t, \mathsf{timeDeadline} : t_1, \mathsf{maxBudget} : n\rangle \ \mathsf{rest},\\
&\mathsf{b} \wedge \mathsf{b}_I) \ \mathsf{if} \ (st = \mathsf{waiting} \vee st = \mathsf{executing})
\end{aligned}$$

where rest is the specification of the remaining tasks, Ats are other attributes of the server, b_I is the set of constraints $t \geq 0 \wedge t_1 \geq 0 \wedge n > 0 \wedge (n - t) > t_1$. This rule specifies that the deadline is missed if there is a task id_0 that is not finished, i.e., either waiting or executing, such that the time to finish (t_1) cannot be met by the available time budget $n - t$ required by the task.

The verification problem of checking whether for some given configuration $(\mathsf{t}_0, \mathsf{b}_0)$ of server and tasks, a task can miss its deadline is specified by the following search command which is an incremental reachability problem

$$\mathsf{search}(\mathsf{t}_0, \mathsf{b}_0) \Rightarrow (\langle \mathsf{id}_1 : \mathsf{global} \mid \mathsf{deadlineMiss} : \mathsf{true}, \mathsf{Ats}\rangle \ \mathsf{rest}, \mathsf{b}') \ \mathsf{such\ that}\ \mathsf{SMT}(\mathsf{b}')$$

Example 2. Rewriting Modulo SMT has been used for verifying whether resource bounded intruders can slowly deny access to webservers [16]. This type of attack was inspired by application layer DDoS attacks such as Slowloris [7] where the attacker attempts to exhaust all the resources of a webserver by periodically sending bursts of multiple requests. When receiving such bursts of requests, the webserver has to allocate resources for at least some period of time, called timeout. As the webserver has limited resources, the attacker is capable of denying service to legitimate users by sending enough bursts.

Constraints were used in previous work [16] to keep track of (1) the number of resources available by the webservers, and (2) the timeout period of bursts. While we refer to the previous work [16] for the complete formalization, we illustrate the incrementality of such specifications with a simplified version of the protocol initialization rule from reference [16].

$$\begin{array}{l}(\mathsf{[iid \mid pxs \mid ri \mid Trec]\ [sid \mid pxs' \mid rs], b}) \rightarrow \\ (\mathsf{[iid \mid px(num, rp)\ pxs \mid ri^{\nu} \mid Trec]\ [sid \mid px(num, rp)\ pxs' \mid rs^{\nu}], b} \wedge \mathsf{b}_I)\end{array}$$

This rule specifies that the intruder iid with ri resources creates a new burst of protocol session instances $\mathsf{px(num, rp)}$ with num instances each using rp resources, where num is a symbol. These instance requests are received by the server sid which has rs resources. The resources of the intruder, ri, and the resources of the server rs are updated to the fresh symbols ri^{ν} and rs^{ν}. These symbols are constrained by the boolean increment b_I defined as $\mathsf{ri}^{\nu} = (\mathsf{ri} - \mathsf{num} \times \mathsf{rp}) \wedge \mathsf{rs}^{\nu} = (\mathsf{rs} - \mathsf{num} \times \mathsf{rp}) \wedge \mathsf{num} > 0 \wedge \mathsf{ri}^{\nu} \geq 0$. Similar rules specify when the protocol sessions timeout and are cleaned up by the server thus releasing resources.

The verification property is to check whether a bounded intruder with some limited number of resources ri can deny service by consuming the server sid's resources. This can be expressed by an incremental reachability property as follows where $(\mathsf{t}_0, \mathsf{b}_0)$ specifies the initial condition when all intruder and server resources are free:

$$\mathsf{search}(\mathsf{t}_0, \mathsf{b}_0) \Rightarrow (\mathsf{[iid \mid pxs \mid ri \mid Trec]\ [sid \mid pxs' \mid rs], b'})\ \mathsf{such\ that\ SMT}(\mathsf{b}' \wedge \mathsf{b}_I)$$

where b_I is the constraint $\mathsf{rs} \leq 0$ specifying that the resources of the server sid are depleted.

Example 3. This example of verification of cyber-physical systems (CPSes) is based on reference [13]. A CPS is represented by a set of agents $(\mathsf{ag}_1, \ldots, \mathsf{ag}_n)$ that interact with the environment (env) to achieve some goal while not violating properties, such as the minimum distance to other objects.

Constraints are used to specify agent's physical attributes, such as its position, $\mathsf{at}(\mathsf{ag}, (x, y))$, speed, $\mathsf{spd}(\mathsf{ag}, v)$, acceleration, $\mathsf{acc}(\mathsf{ag}, acc)$, and direction $\mathsf{dir}(\mathsf{ag}, dir)$ of an agent ag. The evolution of a system with one agent can be specified by the following incremental rule when assuming, for simplicity, that the agent's direction is on the x-axis.

$$\begin{array}{l}([\mathsf{env} \mid \mathsf{at}(\mathsf{ag}, (x, y)), \mathsf{spd}(\mathsf{ag}, v), \mathsf{acc}(\mathsf{ag}, acc), \mathsf{dir}(\mathsf{ag}, dir), \mathsf{kb}]\ \mathsf{conf}, \mathsf{b}) \rightarrow \\ \qquad ([\mathsf{env} \mid \mathsf{at}(\mathsf{ag}, (x_1, y_1)), \mathsf{spd}(\mathsf{ag}, v_1), \mathsf{acc}(\mathsf{ag}, acc), \mathsf{dir}(\mathsf{ag}, dir), \mathsf{kb}]\ \mathsf{conf}, \mathsf{b} \wedge \mathsf{b}_I)\end{array}$$

Here kb is the set of other knowledge-base elements, conf contains the agent's internal representation, x_1, y_1, v_1 are fresh symbols and b_I is set of constraints: $x_1 = (x + (v + v_1) \times \mathsf{dt}/2) \wedge y_1 = y \wedge v_1 = v + \mathsf{acc} \times \mathsf{dt}$. These constraints specify the agent's new position and speed using classical physics equations.

The verification property bad where an agent is too close to an obstacle, such as a pedestrian, is specified by the search command:

$$\begin{array}{l}\mathsf{search}(\mathsf{t}_0, \mathsf{b}_0) \Rightarrow \\ \quad ([\mathsf{env} \mid \mathsf{at}(\mathsf{ag}_1, (x_1, y_1)), \mathsf{at}(\mathsf{ag}_2, (x_2, y_2)), \mathsf{kb}]\ \mathsf{conf}, \mathsf{b}')\ \mathsf{such\ that}\ \mathsf{SMT}(\mathsf{b}' \wedge \mathsf{b}_I)\end{array}$$

where b_I is the set of constraints: $x_1 = x_2 \wedge y_1 = y_2$, specifying that two agents ag_1 and ag_2 are in the same position, i.e., colliding.

3 Hybrid BFS-DFS Algorithm

The definition of Incremental Rewrite Theories addresses the problem of the **Definition of Increments** discussed above. The second problem (**Not possible to chain incremental calls**) still needs to be addressed. Indeed, BFS procedures do not enable the chaining of incremental calls. To illustrate this, consider again the search tree and BFS execution in Fig. 1. Assume that $\mathsf{b}_1 = \mathsf{b}_0 \wedge \mathsf{b}_{0,1}, \mathsf{b}_2 = \mathsf{b}_0 \wedge \mathsf{b}_{0,2}$ and that $\mathsf{goalCond}(\mathsf{t}, \mathsf{b})$ has the form $\mathsf{b} \wedge \mathsf{b}_I$ as one would expect when using incremental rewrite theories. It is possible to call the SMT solver incrementally during the sequence of calls $\mathsf{SMT}(\mathsf{b}_1)$ and $\mathsf{SMT}(\mathsf{goalCond}(\mathsf{t}_1, \mathsf{b}_1))$, but not chain incrementally the call $\mathsf{SMT}(\mathsf{b}_2)$. This is because it is not possible to define an increment between b_1 and b_2 as they lie in different branches of the search tree.

The first obvious alternative is using Depth-First Search (DFS) instead of BFS. This would indeed lead to an execution that could chain incremental calls to the SMT solver. For example, in the tree depicted in Fig. 1, the sequence of calls would be

$$\begin{array}{l}\mathsf{SMT}(\mathsf{b}_0); \mathsf{SMT}(\mathsf{goalCond}(\mathsf{t}_0, \mathsf{b}_0)); \mathsf{SMT}(\mathsf{b}_1); \mathsf{SMT}(\mathsf{goalCond}(\mathsf{t}_1, \mathsf{b}_1)); \\ \mathsf{SMT}(\mathsf{b}_3); \mathsf{SMT}(\mathsf{goalCond}(\mathsf{t}_3, \mathsf{b}_3)) \ldots\end{array}$$

Since b_3 is of the form $\mathsf{b}_0 \wedge \mathsf{b}_{0,1} \wedge \mathsf{b}_{1,3}$, we know the increment is $\mathsf{b}_{1,3}$. There are, however, two problems with DFS. The first problem is that DFS may not find a solution that could be found using BFS due to an infinite branch. The second problem is that the sequence of call using $\mathsf{goalCond}(\mathsf{t}, \mathsf{b})$ appears in between the increments, e.g., $\mathsf{SMT}(\mathsf{b}_0); \mathsf{SMT}(\mathsf{goalCond}(\mathsf{t}_0, \mathsf{b}_0)); \mathsf{SMT}(\mathsf{b}_1)$.

We propose the algorithm $\mathsf{hybrid\_search}$ described in Fig. 2 that addresses these two problems of DFS by combining BFS and DFS and using the PUSH and POP features of SMT solvers for incremental solving. These features enable the creation of backtracking scopes of learned clauses. By default, sequential calls to SMT will attempt to use incremental solving based on the constraints solved in previous calls. A call to PUSH will add to the solver stack any learned clauses from calls to SMT while a call to POP will remove any learned clauses since the last PUSH.

```
1: Queue : FIFO Queue
2: Solver : SMT Solver
3: found : Node
4: function hybrid__search(tree, depth, goal)
5:     found ← NULL
6:     push root of tree on Queue
7:     while Queue has elements and found is NULL do
8:         node ← Queue.pop()
9:         dfs__bounded(node, depth, goal, 0)
10:     end while
11: end function
12: function dfs__bounded(node, max__depth, goal, curr__depth)
13:     b ← node.getBoolean()
14:     Solver.push()
15:     rsat ← Solver.check(b)
16:     if rsat is UNSAT then
17:         Solver.pop()
18:         return
19:     end if
20:     if goal(node) then
21:         found ← node
22:         return
23:     end if
24:     if curr__depth = max__depth then
25:         for all child ∈ node.children() do
26:             Queue.add(child)
27:         end for
28:         return
29:     end if
30:     for all child ∈ node.children() do
31:         dfs__bounded(child, max__depth, goal, curr__depth+1)
32:         Solver.pop()
33:     end for
34: end function
```

Fig. 2. Pseudo-code of the Hybrid Search Algorithm hybrid_search.

The hybrid_search algorithm takes as input the search tree T[3], a non-negative natural number d, and a goal condition g. Intuitively, the parameter d specifies the depth to which the algorithm shall perform DFS before switching to BFS.

We start with *Queue* empty and a *Solver*. hybrid_search starts at line 4 with the next few lines initializing *found* to be NULL and pushing the root of T onto *Queue*. The while loop starts with line 7 continuing while *Queue* is non empty and no solution has been found. It pops the next node off the *Queue* on line 8, then calls dfs_bounded on the next line using this node as the root starting on line 12. dfs_bounded is a modified depth-bounded depth-first search. It starts

[3] Notice that in practice, there is a mechanism that constructs the tree on the fly.

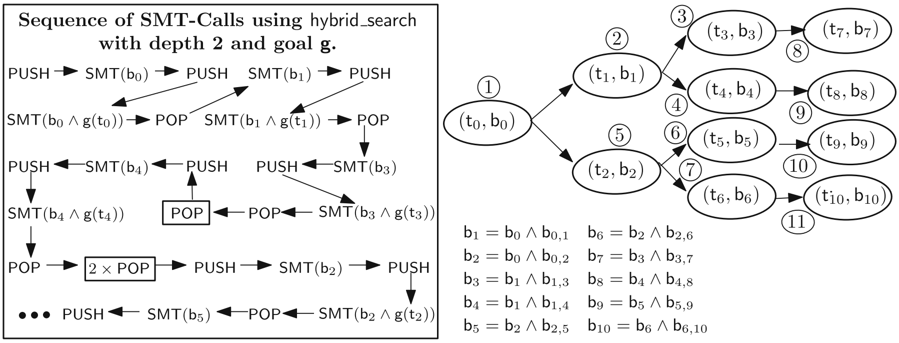


Fig. 3. Illustration of an hybrid_search algorithm execution using the goal condition g and depth two. The POP surrounded by a box indicates the points when the algorithm back-tracks in the search tree. The numbers inside the circles specify the order in which nodes are traversed.

with creating a backtracking scope on *Solver* by calling PUSH and storing the result $\mathsf{SMT}(b)$ where b is the boolean constraint of the current node.

Subsequently, in line 16, it checks if $\mathsf{SMT}(b)$ returned UNSAT, and if so, we POP and return immediately and not explore any children of this node. Any descendent nodes would have a boolean constraint of the form $b \wedge b_I$ for some b_I, and since $\mathsf{SMT}(b)$ is UNSAT it must be the case that $b \wedge b_I$ is also UNSAT. Otherwise, we continue with checking if $goal(node)$ is true on line 20 and if so setting *found* to this node and then terminating dfs_bounded and hybrid_search. If *found* is not set, then line 24 checks when the current depth is equal to the depth parameter d and if it is we add all of the children nodes, i.e., all the nodes that are $d+1$ depth away from the initial root node called from line 9, to *Queue* and no more nodes at a lower depth are visited for now. After all such nodes are added, the execution returns to line 7 to start another dfs_bounded from the next element in *Queue*. Until then, it continues traversing the tree in a DFS-like manner on line 30 ensuring that when dfs_bounded backtracks, we call POP for each node, and hence it backtracks such that *Solver* can properly unlearn clauses that it no longer needs.

We illustrate the execution of hybrid_search with the tree shown in Fig. 3. It also contains the sequence of calls to PUSH, POP and SMT due to the initial call to dfs_bounded. The sequence of calls illustrates the chaining of incremental calls to the SMT solver. For example, the data-structures constructed in the call $\mathsf{SMT}(\mathsf{b}_1)$ are used in the SMT calls for $\mathsf{b}_3, \mathsf{b}_4$, including the calls $\mathsf{goal}(\mathsf{b}_3)$ and $\mathsf{goal}(\mathsf{b}_4)$. This makes sense as b_1 is sub-formula of b_3, b_4, $\mathsf{goal}(\mathsf{b}_3)$ and $\mathsf{goal}(\mathsf{b}_4)$. However, the data-structures constructed in the SMT call for $\mathsf{goal}(\mathsf{b}_1)$ are not stored due to the subsequent POP call, as $\mathsf{goal}(\mathsf{b}_1)$ is not necessarily a subformula of b_3, b_4, $\mathsf{goal}(\mathsf{b}_3)$ and $\mathsf{goal}(\mathsf{b}_4)$. The second observation is the combination of DFS and BFS. While the subtree of depth $d = 2$ is traversed, the algorithm removes the data-structures constructed during the call of $\mathsf{SMT}(\mathsf{b}_1)$, indicated by the $2 \times$ POP in Fig. 3, as b_1 is not necessarily a subformula of b_2.

Notice that the depth parameter (d) plays the role of specifying how much incremental solving one is willing to use with the risk of traversing longer a branch of the search tree that may not have a solution. For example, in the tree and execution shown in Fig. 3, the algorithm will traverse the node $(\mathsf{t}_7, \mathsf{b}_7)$ and will call $\mathsf{SMT}(\mathsf{b}_7)$, but without using the data-structures constructed previously for b_3, that is, it will not solve it incrementally.

The following results relate hybrid_search with BFS and with DFS.

Proposition 1. *Let T be a tree and g be a decidable goal condition. Then,* $\mathsf{hybrid\_search}(T, 0, g)$ *will traverse T in the same order as BFS.*

Proof Sketch. A DFS search bounded by depth 0 will only traverse a single node, the node it starts at. Then, it adds nodes to a FIFO queue in the same manner as BFS. Hence, $\mathsf{hybrid\_search}(T, 0, g)$ will traverse T in the same order as BFS. **QED**.

Proposition 2. *Let T be a tree and g be a decidable goal condition. Suppose the depth of T is d. Then, for any $k \geq d$,* $\mathsf{hybrid\_search}(T, k, g)$ *will traverse T in the same order as DFS.*

Proof Sketch. If k is greater than or equal to the depth of T, then a k depth-bounded DFS from the root node would traverse all of T. Hence, $\mathsf{hybrid\_search}(T, k, g)$ traverses the T in the same order as DFS. **QED**.

The following statement provides coverage guarantees.

Proposition 3. *Let $d > 0$, T be a tree of finite branching, and g be a decidable goal condition. Then,* $\mathsf{hybrid\_search}(T, d, g)$ *finds a solution in finite time, i.e., some node n in T such that $g(n)$ is true, if such a solution exists.*

Proof. Let B_i be the number of nodes in T at depth i. Suppose that the solution node n exists at depth r and no solutions exist at a lower depth. Let $0 \leq r \leq qd$ for some q. The first depth-bounded DFS will traverse all nodes up to depth d. This then adds B_{d+1} nodes to *Queue*. Running the depth-bounded DFS run these nodes will traverse all the nodes to $2d$. Traversing all nodes up to qd would take $1 + B_{d+1} + B_{d+2} + ... + B_{qd}$ iterations of depth-bounded depth first searches. Since n exists at depth $r \leq qd$ and each B_i is finite since T has finite branching, n would be found in finite time. **QED**.

To address the fact that search trees may have infinite depth, often one uses bounded search that searches the tree until only some given depth d. The following proposition states that in these cases it is best to deploy hybrid_search with depth d to search through all nodes of the sub-tree, provided incremental SMT calls are more efficient than SMT calls from scratch.

Proposition 4. *Let T be a tree of finite branching with branching factor b and g be a decidable goal condition. Let $T(d)$ be the sub-tree of T of depth d with $d > 0$. Assume that incremental SMT calls, i.e., using* PUSH, *take less time than calls from scratch, i.e., without using* PUSH. *Then for any $d' \geq 0$ such that $d' \neq d$, the time required by* $\mathsf{hybrid\_search}(T, d, g)$ *to traverse all nodes in $T(d)$ is less than the time of* $\mathsf{hybrid\_search}(T, d', g)$ *to traverse all nodes in $T(d)$.*

Proof. Let $0 < r < 1$ be the average performance benefit from incremental SMT calls and t be the time it takes for non-incremental SMT calls. Let B_i be the number of nodes at depth i. Since b is finite, each B_i is finite. The time required by hybrid_search(T, d, g) to traverse all nodes in $T(d)$ is $t + rtB_1 + rtB_2 + ... + rtB_d$. Suppose that $0 < d' < d$. Let $pd' < d \leq (p+1)d'$ for some p. For hybrid_search(T, d', g) to traverse all nodes in $T(d)$, it must traverse all nodes in $T((p+1)d')$ because each dfs_bounded must travel exactly d' depth, hybrid_search(T, d', g) will traverse only depths that are multiples of d'. Then, the time required for hybrid_search(T, d', g) is $t + rtB_1 + ... + rtB_{d'} + tB_{d'+1} + rtB_{d'+2} + ...rtB_{2d'} + ... + tB_{pd'} + rtB_{pd'+1} + ... + rtB_{(p+1)d'}$. There are $p+1$ terms that do not get the benefit from incremental SMT calls for hybrid_search(T, d', g) while there is 1 term that does not get this benefit for hybrid_search(T, d, g). Hence, the time required for hybrid_search(T, d, g) to traverse all nodes in $T(d)$ is less than the time required for hybrid_search(T, d', g) to traverse all nodes in $T(d)$. Now, suppose that $d' > d$. Then, for hybrid_search(T, d', g) to traverse all nodes in $T(d)$, it must traverse all nodes in $T(d')$. The time required for hybrid_search(T, d', g) is $t + rtB_1 + rtB_2 + ... + rtB_{d'}$. But, because $d' > d$ and each $rtB_i > 0$ the time required for hybrid_search(T, d, g) is less than hybrid_search(T, d', g). Hence, the time required for hybrid_search(T, d, g) to traverse all nodes in $T(d)$ is less than the time required for hybrid_search(T, d', g) to traverse all nodes in $T(d)$. Therefore, for any $d' \neq d$ the the time required for hybrid_search(T, d, g) to traverse all nodes in $T(d)$ is less than the time required for hybrid_search(T, d', g) to traverse all nodes in $T(d)$. **QED**.

4 Implementation and Experiments

Our implementation is based on Python with the Z3 SMT solver and Maude integrated using Python bindings [15] as depicted in Fig. 4. The Z3 Solver is responsible for checking the incremental satisfiability of constraints using SMT, PUSH and POP, while Maude is responsible for executing rewriting rules. The Maude bindings allow for loading Maude files into the Python implementation of hybrid_search. The search is done with a Python function that repeatedly calls the Maude search with one step (Search1) so that the traversal of the search space can be controlled. The original Maude specifications were modified to replace calls to SMT with calls to functions defined using the Maude hook mechanism for attaching external code to function symbols. This mechanism is exposed by the Maude Python bindings. There are two types of function, one that checks satisfiability while keeping any learned clauses from the check, and one that just checks without adding any learned clauses. The functions keep track of the SMT solver state using appropriate calls to PUSH and POP. The implementation is available at [17].

Figures 5, 6 and 7 summarize the experiments carried out using implementations available in the literature [3,13,16] for the verification of the systems described in Examples 1, 2, and 3. All experiments were run on a Windows 10 machine, Intel Core i7-10700J, 16 GB of RAM, on Python 3.10.2, using Maude

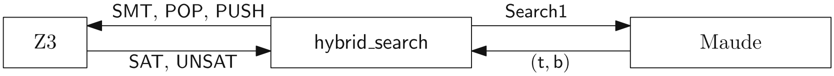


Fig. 4. Overview of the implementation used for the experiments using hybrid_search, the SMT solver Z3 and the rewriting tool Maude.

Python bindings 1.1.2 and Z3 4.11.2.0. We measure the runtime for these three applications of rewriting modulo SMT to determine the performance gain from using hybrid search at various depth parameters compared to BFS and DFS. Each table shows the initial configuration for the system, then statistics for searches for BFS, DFS, and using hybrid_search at various depths terminating when finding a single goal node. The statistics have the form $n/m/p$ which specify the time n in seconds to perform verification, the number of states m traversed, and the percentage p of verification time required by SMT-solving. DNF indicates that no solution was found within 30 min. For example, the first row for cashOK_1 using the BFS mechanism for instance, the execution time was 6.9 seconds, requiring 91 state traversals while spending 77% of execution time in Z3.

For our experiments, we used the same subsets of the verification problems used in references [3,13,16]:

- $\mathsf{cashOK}(I_0, I_1, I_2, I_3, \mathsf{b})$ and $\mathsf{cashBad}(I_0, I_1, I_2, I_3, \mathsf{b})$ correspond to symbolic initial configurations of a CASH scheduling problem with two servers (see Example 1). I_0 and I_1 specify, respectively, the maximum budget and the period of the first server, while I_2 and I_3 specify, respectively, the maximum budget and period of the second server. b is a constraint on the values of I_1, I_2, I_3, and I_4. cashOK uses a correct implementation of the scheduler, while cashBad uses an incorrect specification.
- $\mathsf{Slowloris}(P_1, P_2, \mathsf{DoSDur})$ corresponds to symbolic initial configurations of a Slowloris verification problem (see Example 2). P_1 specifies the bound on the number of parallel bursts of symbolic protocols, and P_2 specifies the bound on the number of different types of messages sent in parallel, where $P_2 = 0$ denotes no bound. Moreover, DoSDur specifies the minimum duration for which the server's resources are depleted in order to consider the DoS attack successful.
- $\mathsf{pedestrian}(t, \mathsf{Safer}, \mathsf{Safe}, \mathsf{Unsafe})$ specifies a pedestrian crossing scenario problem where an autonomous vehicle is approaching a pedestrian crossing. The verification problem is to avoid an unsafe situation. The three levels of safety are defined according to the parameters $\mathsf{Safer} > \mathsf{Safe} > \mathsf{Unsafe}$ specifying bounds on the distance to between the vehicle and the pedestrian measured in terms of time to travel. The verification problem is to determine whether a given vehicle controller cannot reach an unsafe situation within t time units when starting at a safe situation. The size of a time unit is $0.1s$.

The results for the CASH verification experiments show that hybrid_search finishes up to about 10 times faster than BFS and terminates in all cases as

Init	BFS	DFS	HYBRID d=2	HYBRID d=4	HYBRID d=8
cashOK$_1$	6.9 / 93 / 77%	0.7 / 9 / 8%	**0.2 / 37 / 21%**	1.3 / 117 / 12%	0.7 / 9 / 8%
cashOK$_2$	8.0 / 100 / 71%	3.0 / 12 / 4%	**0.6 / 52 / 13%**	1.9 / 118 / 9.0%	0.9 / 14 / 6%
cashOK$_3$	3.5 / 65 / 84%	DNF	0.1 / 32 / 25%	**0.06 / 9 / 26%**	1.3 / 28 / 6%
cashBad$_1$	5.9 / 63 / 74%	0.7 / 9 / 7%	**0.2 / 27 / 21%**	1.2 / 61 / 10.%	0.7 / 9 / 8%
cashBad$_2$	7.5 / 70 / 69%	2.9 / 12 / 4%	**0.6 / 42 / 13%**	1.8 / 62 / 7.7%	0.9 / 14 / 6%
cashBad$_3$	2.6 / 39 / 81%	DNF	**0.1 / 22 / 26%**	0.06 / 9 / 24%	1.4 / 28 / 6%

Fig. 5. CASH Verification Experiments. cashOK$_1$ = cashOK$(I0, I1, I2, I3, true)$, cashOK$_2$ = cashOK$(I0, I1, I2, I3, I0 + I3 > I1 + I2)$, and caseOK$_3$ = caseOK$(I0, I1, I2, I1, I0 + I2 > I1)$, and *mutatis mutandis* for cashBad$_1$, cashBad$_2$ and cashBad$_3$.

Init	BFS	DFS	HYBRID d=2	HYBRID d=3	HYBRID d=4
Slow$_1$	2.4 / 51 / 88%	0.2 / 66 / 39%	0.3 / 52 / 56%	0.4 / 79 / 57%	**0.2 / 35 / 41%**
Slow$_2$	39.8 / 775 / 83%	DNF	8.0 / 1612 / 44%	**3.1 / 703 / 39%**	13.0 / 3314 / 36%
Slow$_3$	0.5 / 11 / 87%	0.06 / 10 / 50%	0.06 / 9 / 46%	**0.05 / 8 / 52%**	0.06 / 9 / 50%
Slow$_4$	1.8 / 29 / 86%	0.1 / 27 / 39%	0.2 / 27 / 55%	0.2 / 34 / 54%	**0.1 / 20 / 41%**
Slow$_5$	19.0 / 147 / 78%	DNF	2.5 / 187 / 44%	1.3 / 118 / 41%	3.9 / 261 / 39%

Fig. 6. Slowloris Experiments. Slow$_1$ = Slowloris$(1, 0, 24)$, Slow$_2$ = Slowloris$(1, 0, 36)$, Slow$_3$ = Slowloris$(1, 1, 12)$, Slow$_4$ = Slowloris$(1, 1, 24)$, Slow$_5$ = Slowloris$(1, 1, 36)$.

Init	BFS	DFS	HYBRID $d = t$	HYBRID $d = 2 \times t$	HYBRID $d = 3 \times t$
cps$_1$	12.3 / 119 / 62%	**4.8 / 57 / 68%**	7.0 / 117 / 65%	6.6 / 99 / 67%	5.0 / 57 / 71%
cps$_2$	53.4 / 323 / 71%	23.0 / 152 / 78%	**15.6 / 213 / 69%**	28.6 / 232 / 77%	22.7 / 152 / 78%
cps$_3$	301.0 / 819 / 84%	97.3 / 387 / 85%	63.9 / 429 / 80%	**52.7 / 364 / 82%**	99.7 / 387 / 85%
cps$_4$	12.5 / 119 / 63%	4.8 / 57 / 70%	7.8 / 118 / 69%	6.4 / 99 / 66%	**4.7 / 57 / 68%**
cps$_5$	56.0 / 323 / 72%	25.4 / 152 / 80%	**18.4 / 227 / 71%**	19.8 / 192 / 74%	23.3 / 152 / 79%
cps$_6$	285.1 / 819 / 83%	100.9 / 387 / 85%	**60.2 / 424 / 79%**	84.6 / 437 / 85%	101.4 / 387 / 85%

Fig. 7. Cyber-Physical System Verification Experiments, where cps$_1$ = pedestrian$(3, 3, 2, 1)$, cps$_2$ = pedestrian$(4, 3, 2, 1)$, cps$_3$ = pedestrian$(5, 3, 2, 1)$, cps$_4$ = pedestrian$(3, 4, 2, 1)$, cps$_5$ = pedestrian$(4, 4, 2, 1)$, cps$_6$ = pedestrian$(5, 4, 2, 1)$. The bound t, $2 \times t$ and $3 \times t$ is determined according to the t parameter of the scenario.

opposed to two of the DFS cases where it does not finish within 30 min. The overhead of Z3 is reduced from about 70% to 80% down to 6% to 25% from BFS to hybrid_search. This indicates the effectiveness of the incremental SMT solving for the types of constraints used in this example.

Similarly, in the Slowloris examples, hybrid_search finishes up to 10 times faster than BFS with termination while two of the DFS cases do not finish within 30 min. In these cases the overhead of Z3 goes from about 80% to 90% in BFS while it goes from about 30% to 60% in hybrid_search, demonstrating the effectiveness of the incremental solving. Interestingly, even when there is a much larger number of states traversed, e.g., in case Slow_2 and HYBRID d = 4 with 3314 states traversed as opposed to 775 states traversed by BFS, the verification time is one third, from about 40s to 13s. This indicates that the main overhead of BFS is indeed SMT solving.

For the Cyber-Physical System (CPS) Verification experiments, hybrid_search completes up to about 5 times faster than BFS. The overhead of Z3 does not change significantly in these experiments, which indicates that the incremental solving is not as effective as in the other two examples (CASH and Slowloris). The reason for this may be the non-linear nature of the constraints for CPS systems which contrast with the former two examples that use linear arithmetic constraints. Despite this, hybrid_search and DFS still outperform BFS because they need to traverse fewer nodes before finding a goal node.

5 Related Work

We consider three related areas of work in optimizing symbolic execution modulo SMT, hybrid search strategies, incremental constraint solving methods, and tradeoffs between search space and constraint complexity.

Hybrid Search Strategies. There have been others that have previously explored techniques of combining BFS and DFS so to take advantage of both of their benefits while reducing the drawbacks of each.

Reference [5] proposes a hybrid algorithm for Binary Decision Diagrams (BDDs). BDDs are are often used to represent and manipulate boolean functions symbolically. Traditionally, depth-first approaches were used in the construction of BDDs as it had relatively low memory overhead. Though, it had been discovered that using a breadth-first approach instead had better performance due to better memory access locality at the cost of larger memory overhead. To improve upon both approaches a hybrid of the two is used. Essentially, the algorithm switches between the two techniques based on its memory overhead. When the memory overhead is computed to be low, a breadth-first search is used and when it is high a depth-first search is used.

Reference [1] constructs a "breadth-first, depth-next" algorithm for building Random Forest (RF) models. An RF model is a machine learning model that uses decision trees. Both DFS and BFS approaches are used in machine learning frameworks. They observe that BFS has memory efficient access patterns at lower depths. As the depth increases it loses this benefit and virtually has random access to memory. At this point, DFS performs better. As a result, their algorithm starts with a breadth-first approach until it is computed that is no longer has efficient access pattern, switching to a depth-first approach.

Reference [9] introduces "depth-first iterative-deepening (DFID)." One of the issues with BFS is that it has exponential memory complexity. DFS can circumvent this drawback as its memory complexity is linear, but comes with its own problems. It generally requires some depth bound and check for repeated nodes, otherwise the search may not terminate. The actual depth bound needed may not be knowable at runtime and choosing a bound too low may result in the search ending without finding the solution. To counteract the downsides of BFS and DFS, DFID is used. DFID starts with DFS bounded by depth one, then performs a DFS bounded by depth two, and continue this process with incrementally larger bounded depths until a solution is found. It must visit the same nodes multiple times, but it is shown that the runtime complexity is not effected by it.

Unfortunately, none of these algorithms seem particularly helpful with respect to rewriting modulo SMT. For example, prior algorithms [1,5] attempt to take advantage of memory locality as much as possible. In our case, it would not give us much performance increase. Reference [9] requires nodes to be visited multiple times. This would lead to duplicate calls the SMT solver, only increasing the bottleneck.

Incremental Solving. In reference [10] the authors compare cache-based and stack-based incremental constraint solving methods in the context of symbolic execution for test generation. Cached-based incrementality works outside the solver to cache results and attempt to reuse them. Stack-based incrementality uses a solvers ability to reuse information learned when solving a subproblem and the associated push/pop interface. Implementations of the two methods and a baseline (no incrementality) were compare on large benchmark set of C programs and on randomly generated programs. The space of symbolic execution paths was searched using bounded depth first search. The authors found that caching generally increased average solving time over baseline (by a factor of 2–5 depending on code size), while stack-based methods decreased average solving time by roughly a factor of 20. This is consistent with our observations even though the source of search tree is different and the class of constraints is different.

Trading Search Space for Constraint Complexity. A notion of guarded term is introduced in reference [2] as a method to reduce the search state space in symbolic rewriting modulo SMT by replacing non-determinism by disjunction. The effect of using guarded terms is demonstrated in a study of the CASH algorithm for task scheduling. Many properties that could not be checked using symbolic execution modulo SMT (due to size of search space and timeout) became tractable using guarded terms.

A study of the tradeoff between search space size and constraint size using symbolic execution modulo SMT in the context of analyzing safety of autonomous systems such as platooning scenarios is presented in reference [13]. The results in that paper suggest that not only the size of state space matters for automation, but also the size of constraints that are sent to the SMT solver as many searches fail to terminate due to non-termination of constraint solving when constraints get large, while the same searches terminate with disjunctions are turned into branching in the search space.

None of these approaches, however, investigate the use of incremental SMT solving for improving performance of Rewriting Modulo SMT.

6 Conclusions and Future Work

This paper proposes Incremental Rewrite Theories that enable incremental SMT solving for rewriting modulo SMT. This is accomplished by the search procedure hybrid_search which combines BFS and DFS. The effectiveness of hybrid_search is demonstrated by using a collection of verification problems taken from the literature, including algorithm verification, network security analysis, and cyber-physical systems safety verification. In all examples, the time taken to verify by hybrid_search improved by a factor between 5–10 when compared to traditional BFS approaches, showing the great benefits of using incremental solving.

The current notion of incremental rewrite theory is essentially a syntactic notion although equational theories are used to reduce terms and matching may be modulo axioms, such as associativity and commutativity. This makes identifing the boolean increment efficient and thus well suited for the hybrid algorithm. An interesting direction for future work is to investigate less restrictive notions of *incremental* and indentify more general classes of rewrite theories where incremental solving is effective. Another direction of future work we are investigating is the trade-offs of incremental solving and the shape of constraints, e.g., use disjunctions to reduce search space versus split disjunctions to reduce SMT solving time. We also are investigating the incorporation of incremental solving algorithms in tool implementations such as Maude.

Acknowledgments. This research is funded in part by the Ashton Fellowship at the University of Pennsylvania. Talcott was partially supported by the U. S. Office of Naval Research under award number N00014-20-1-2644, and NRL grant N0017317-1-G002. We also thank the anonymous reviewers for their careful reading and insightful comments.

References

1. Anghel, A., Ioannou, N., Parnell, T., Papandreou, N., Mendler-Dünner, C.: Breadth-first, depth-next training of random forests. In Neural Information Processing Systems (NeurIPS) (2019)
2. Bae, K., Rocha, C.: Symbolic state space reduction with guarded terms for rewriting modulo SMT. Sci. Comput. Program. **178**, 20–42 (2019)
3. Bae, K., Rocha, C.: Symbolic state space reduction with guarded terms for rewriting modulo SMT. In: Formal Aspects of Component Software (FACS) (2019)
4. Caccamo, M., Buttazzo, G.C., Sha, L.: Capacity sharing for overrun control. In: Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000), pp. 295–304. IEEE Computer Society (2000)
5. Chen, Y., Yang, B., Bryant, R.: Breadth-First with Depth-first BDD Construction: A Hybrid Approach (1997)
6. Clavel, M., et al.: All About Maude: A High-Performance Logical Framework, vol. 4350. Springer, Heidelberg (2007)
7. Dantas, Y.G., Nigam, V., Fonseca, I.E.: A selective defense for application layer DDoS attacks. In: IEEE Joint Intelligence and Security Informatics Conference, JISIC 2014, pp. 75–82. IEEE (2014)

8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
9. Korf, R.: Depth-First Iterative-Deepening: An Optimal Admissible Tree Search (1985)
10. Liu, T., Araújo, M., d'Amorim, M., Taghdiri, M.: A comparative study of incremental constraint solving approaches in symbolic execution. In: Yahav, E. (ed.) Hardware and Software: Verification and Testing. pp, pp. 284–299. Springer International Publishing, Cham (2014)
11. MaudeSE (2021). https://github.com/maude-se/maude-se.github.io
12. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992)
13. Nigam, V., Talcott, C.: Automating safety proofs about cyber-physical systems using rewriting modulo SMT. In: Bae, K. (ed.) WRLA 2022. LNCS, vol. 13252, pp. 212–229. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-12441-9_11
14. Rocha, C., Meseguer, J., Muñoz, C.: Rewriting modulo SMT and open system analysis. J. Logical Algebraic Methods Program. **86**, 269–297 (2017)
15. Rubio, R.: Maude as a library: an efficient all-purpose programming interface. In: Bae, K. (ed.) WRLA 2022. LNCS, vol. 13252, pp. 274–294. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-12441-9_14
16. Urquiza, A.A., et al.: Resource and timing aspects of security protocols. J. Comput. Secur. **29**(3), 299–340 (2021)
17. Whitters, G., Nigam, V., Talcott, C.: Incremental rewriting modulo SMT experiments (2023). https://github.com/WhittersGerald/cade-incremental-rewriting

Iscalc: An Interactive Symbolic Computation Framework (System Description)

Bohua Zhan[1,2](✉), Yuheng Fan[3], Weiqiang Xiong[2], and Runqing Xu[1]

[1] State Key Laboratory of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing, China
[2] University of Chinese Academy of Sciences, Beijing, China
bzhan@ios.ac.cn
[3] National Computer System Engineering Research Institute of China,
Beijing, China

Abstract. The need to verify symbolic computation arises in diverse application areas. In this paper, based on earlier work on verifying computation of definite integrals in HolPy, we present a tool Iscalc for performing a variety of symbolic computations interactively, taking a middle ground in terms of easy of use and rigor between computer algebra systems and interactive theorem provers. The tool supports user-level definitions and dependency among computations, allowing construction and reuse of custom theories. Side conditions are checked on a best-effort basis. The tool is applied to highly non-trivial computations from the textbook Inside Interesting Integrals.

Keywords: Symbolic computation · User interface · Computer algebra

1 Introduction

Symbolic computations arise in many mathematical proofs as well as in science and engineering. The use of computers to ensure their correctness is hence an important problem. Interactive theorem provers and computer algebra systems provide two alternative approaches. Most interactive theorem provers have extensive libraries in analysis [6], based upon which one can verify correctness of computations with a very high level of confidence. However, the learning curve for using such libraries is quite steep. On the other hand, computer algebra systems, such as Mathematica, Maple, etc, aim to perform computations automatically. However, it is difficult to guide the computation if the automatic procedure fails, and the correctness is not fully guaranteed. Indeed there have been examples of mistakes made by such computer algebra systems in the past [11].

Previous work [18] introduces a system for performing and verifying symbolic computation as an extension to the HolPy interactive theorem prover [19]. The user can perform calculation of definite integrals step-by-step, using rules

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 577–589, 2023.
https://doi.org/10.1007/978-3-031-38499-8_33

such as substitution, integration by parts, etc. Each step has a relatively simple implementation, and proofs in higher-order logic can be constructed automatically from the sequence of steps, which in turn can be checked by the HolPy kernel. This provides a user experience which can be seen as a mix between the two approaches discussed above, combining the more intuitive feel of computer algebra systems with higher level of confidence in the results.

In this paper, we present a significant extension to the work in [18], forming an independent tool named Iscalc (**I**nteractive **s**ymbolic **calc**ulations). In particular, we make the following extensions aimed at greater safety, extensibility, and ability to handle a wider range of examples.

1. We introduce user-level definitions and dependency among computations, allowing construction and reuse of custom theories. This is achieved by maintaining *contexts*, which contain the list of existing definitions and identities, as well as assumptions in the current computation.
2. We introduce systematic checks on wellformedness of expressions and side-conditions for applying certain rules within Iscalc (rather than only when reconstructing proofs). This increases confidence in the computation without proof reconstruction.
3. In addition to definite integrals, the tool now supports computation with limits, series, and indefinite integrals. We also support improper integrals, and many more techniques of computation, such as series expansions and differentiating under the integral sign.
4. With only few exceptions (such as partial fraction decomposition), all functionalities are now implemented independently rather than depending on SymPy. We found this approach, aimed at avoiding problems caused by limitations of SymPy, to be more flexible and extensible in the end.

One of our main aims and yardstick for measuring progress is verifying computations from the textbook *Inside Interesting Integrals* [17]. This book contains many computations of integrals using a variety of techniques, including differentiating under the integral sign, series expansions, and so on. Many computations are quite involved (the longest example we did, Ahmed's Integral, is 4 pages long in the book). We also carry over and complete some of the case studies in [18].

Our aim is to provide a user interface that is more intuitive and accessible to mathematicians and engineers. In particular, computations are displayed in LaTeX form, and whenever there is tension between conventional mathematical language and the more precise formal language, we prefer the former. We take the best-effort approach to correctness, providing systematic checks for the usual mistakes, such as cancelling expressions that may be zero, or exchange of sums that are not absolutely convergent. However, full correctness guarantees in the sense of interactive theorem proving is not achieved without proof reconstruction, which we leave to future work. In this respect, our approach is more similar to SMT solvers and program verification tools based on them, which sacrifice some correctness guarantees for more efficiency and speed of development.

We now give an outline for the rest of this paper[1]. Section 2 describes the overall architecture of Iscalc. Section 3 shows results of case studies, and gives

[1] Source code and examples are available at https://github.com/bzhan/iscalc.

some interesting examples. Section 4 discusses some lessons we took from this work, especially for user interface design. Section 4.1 discusses related work and Sect. 5 concludes the paper.

2 Architecture

Iscalc has a layered architecture consisting of several modules, as shown in Fig. 1. In this section, we begin with some preliminary definitions, then describe the functionality of each module in turn.

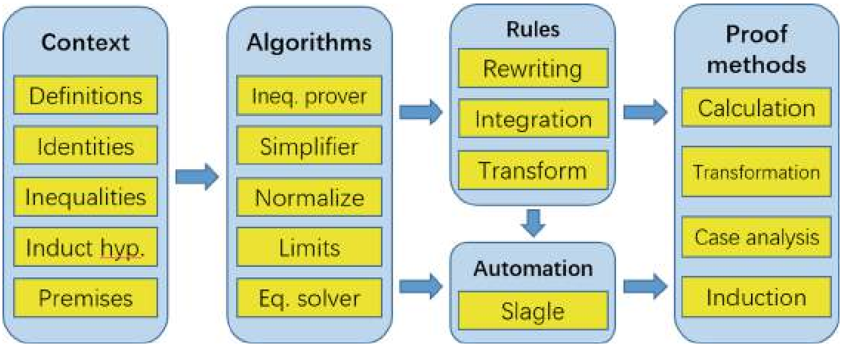


Fig. 1. Overall Architecture

2.1 Preliminaries

The term language of Iscalc inherits from that in [18], but with extensions for limits, summation, and indefinite integrals. The full syntax is as follows.

$$\begin{aligned} e := {} & v \,|\, c \,|\, e_1 \;\, op \;\, e_2 \,|\, f(e) \,|\, \mathsf{Deriv}(e,v) \,|\, \mathsf{Integral}(e,v,a,b) \,| \\ & \mathsf{Limit}(e,v,a,\mathit{dir}) \,|\, \mathsf{Sum}(e,i,a,b) \,|\, \mathsf{IndefiniteIntegral}(e,v,\mathit{deps}) \,|\, \mathsf{Skolem}(n,\mathit{deps}) \end{aligned}$$

Constructors on the first line stand for variables, constants, operators, function applications, derivatives, and definite integrals, respectively. Constants are extended to include positive and negative infinities. Constructors on the second line are new, and we explain them in more detail.

$\mathsf{Limit}(e,v,a,\mathit{dir})$ represents the limit of expression e as variable v goes to expression a, here dir represents the direction of the limit. That is, we distinguish between $\lim_{x\to 0+} f(x)$ and $\lim_{x\to 0-} f(x)$, etc. $\mathsf{Sum}(e,i,a,b)$ represents summation of expression e as the integer index i goes from a to b (inclusive, except when $b=\infty$). $\mathsf{IndefiniteIntegral}(e,v,\mathit{deps})$ and $\mathsf{Skolem}(n,\mathit{dep})$ are used together for computing with indefinite integrals. The former represents indefinite integral of e with respect to v. When this is evaluated to an expression plus "C", this

C is represented by a Skolem term. Here *deps* represent the additional variables that C may depend on, which comes from the list of dependent variables *deps* of the indefinite integral. The use of dependent variables in evaluating indefinite integrals is illustrated by an example in Sect. 3.1.

Another extension compared to [18] is the addition of formulas. These are used to specify goals, wellformedness conditions on terms, as well as assumptions on goals and definitions. Currently we support the following constructors for formulas:[2]

$$f := e_1 \; op \; e_2 \mid \mathsf{isInt}(e) \mid \mathsf{notInt}(e) \mid \mathsf{converges}(e)$$

where the binary operator *op* is one of $=, \neq, <, \leq, >, \geq$. $\mathsf{isInt}(e)$ and $\mathsf{notInt}(e)$ represent e is/is not an integer. $\mathsf{converges}(e)$ represents e is convergent, where e is a series whose upper limit is ∞.

2.2 Context

In [18], each computation is independent from each other, and all available definitions and identities are built into the kernel. In contrast, Iscalc develops a system of user-level definitions and dependency between computations similar to usual interactive theorem provers. This is achieved by a hierarchy of *books*, *files*, *definitions* and *goals*. Each book consists of an ordered list of axioms, definitions, and files, and may depend on other books. Each file contains a list of goals, whose computation may depend on previous items in the book. Each definition specifies a new function along with assumptions on the arguments of that function. Each axiom or goal specifies a single expression to be proved under a set of premises. It may be marked with *attributes* to specify its type or how it is to be used (e.g. whether it can be used during simplification).

In the implementation, a Context object maintains the list of definitions, identities, and inequality rules available at the current file. It also contains the premises and inductive hypothesis for the current computation (these are modified when performing a case analysis or induction, as described in Sect. 2.5).

2.3 Algorithms

Iscalc implements several basic algorithms in computer algebra, for checking inequalities, simplification and normalization of expressions, computing limits, and solving equations. All of these take a Context object as input, and depend on the context information.

[2] Currently we do not use logical operators, as negation is unnecessary for the current list of formulas, and conjunction and disjunction are represented using internal data structures. This may change as new needs arise in the future.

Inequality Checking. Unlike in the previous paper, condition checking is implemented entirely from scratch rather than relying on SymPy. It is well-known that checking inequalities involving transcendental functions is undecidable. Our goal is to perform simple rule-based reasoning automatically, leaving more involved inequalities to be proved with user guidance. The overall approach is saturation: we maintain a dictionary mapping expressions to conditions on them. Given an expression for which we wish to derive some conditions, saturation works recursively on each subexpression, matching it against the main argument of each rule (left side of inequalities, or the last argument of predicates). For each match, it looks in the dictionary for existing facts that justifies assumptions of the rule. Special reasoning is performed on numerical constants (e.g. $x < c_1$ can be used to justify $x < c_2$ if $c_1 \leq c_2$). Comparison between numerical constants are currently done with floating-point approximation.

The approach described here is relatively simple, and it is not difficult to ensure termination, as we only get conditions on expressions that already appear. However, in practice it can be quite powerful when combined with user-guided rewriting, as shown by the example in Sect. 3.2.

Simplification. Simplification of expressions works in mostly the same way as [18], and we restate the main ideas. We normalize with respect to AC-property of addition and multiplication, and combine equal terms. When trying to combine $t^a t^b$ into t^{a+b}, we check using the current context that either t is nonzero and a, b are integers, or t is nonnegative. This prevents cancellation of e.g. t/t into 1 when t may be zero.

Moreover, we apply identities in the context that are marked with the simplify attribute. These cover evaluation of functions at special values, as well as issues like removal of absolute value sign (e.g. $|x| = x$ if $x \geq 0$).

Normalization. There are situations where different forms of an expression are desirable for different purposes, e.g. factorized vs. expanded form of a polynomial, single quotient vs. a sum of quotients, etc. We designed the simplifier to not make a choice in such situations. Instead, if the user wishes to convert an expression to a different form, she can specify the rewriting explicitly. Iscalc then normalizes both old and new expressions and check whether they are equal. Normalization expands polynomials and combines quotients (e.g. for checking partial fraction decomposition), and performs (among others) rewriting of logarithm and exponentials.

Computing Limits. For limit computations, we implement a simplified version of the approach by Gruntz [10]. To compute $\lim_{x \to \infty} e$, we evaluate recursively the limit of each subexpression in e, as well as the asymptotics of approaching that limit. Possible asymptotics include powers of polynomials and logarithms, as well as exponentials. Finding the limit as x approaches other values is converted to computing the limit at infinity.

As with other algorithms, the aim is not to achieve high level of automation, but to perform the simpler limits, leaving more complex cases to human guidance (e.g. using L'Hopital's rule or with rewriting). On the other hand, using the complete algorithm of Gruntz, or the algorithm implemented by Eberl in Isabelle [8], would certainly increase automation and range of applications.

Solving Equations. We implement simple equation solving, including isolating the expression to be solved, and solving linear equations. This is used when performing substitutions and in transforming/applying an existing equality.

2.4 Rules

Based upon the collection of algorithms in the previous section, Iscalc implements a set of rules for transforming the current expression in a computation. Currently 37 rules are available. We give some representative examples below.

Integration Rules. The list of integration rules are mostly inherited from [18]. They include Substitution, IntegrationByParts, etc. Integration identities can be applied by lookup from the context. There are also rules for more advanced techniques such as differentiating under the integral sign (illustrated in Sect. 3.1), and exchange of integral and sum (illustrated in Sect. 3.3).

Rewriting Rules. The most basic rewriting rule is FullSimplify, which applies simplification to the current expression. ApplyIdentity applies an identity from the context. This generalizes the use of Fu's rules for trigonometric identities [9]. The rule Equation supports rewriting to another form of an expression with equal normal form. Series expansion and evaluation of series are available as two different rules (again looking up identities from the context).

Equality Transformation Rules. These rules transform one equality into another. IntegralEquation transforms an equation of the form $\mathsf{Deriv}(e, x) = g(x)$ into $e = \mathsf{IndefiniteIntegral}(g, x, \mathit{fvars})$, where fvars is the list of free variables in $\mathsf{Deriv}(e, x)$. Another very flexible rule is SolveEquation, which solves for some expression e in an equality $s = t$ to give another equality $e = e'$. Other examples include taking limit on both sides, applying a function to both sides, and so on.

Other Rules. Besides the above three major categories, other rules include the L'Hopital's rule for computing limits, and rules for series manipulations.

2.5 Proof Methods

In [18], the only way to perform a computation is starting from a single expression, and applying rules to transform that expression. More complex applications necessitate more structures in the computation. We describe those supported by Iscalc briefly, as they are all familiar from other theorem provers.

Proof by Computation. To show an equality $a = b$, perform computation on both sides until they become identical. Likewise, for inequalities, perform computation on both sides until the inequality can be shown automatically.

Proof by Transformation. Starting from a known equality $a = b$, apply the equality transformation rules in Sect. 2.4 to obtain new equalities, until the desired one is obtained.

Case Analysis. To show a goal, divide into cases either by whether some comparison formula is true, or according to whether some expression is less than, equal to, or greater than 0. We shown an example with inequality goals in Sect. 3.2.

Induction. Some integrals involve an integer parameter $n \geq 0$, and may be proved by induction on n. We support such inductive reasoning in Iscalc. The rule ApplyInductHyp can be used to apply inductive hypothesis at any time in the inductive branch of the proof.

2.6 Top-Level Computation, Automation, and User Interface

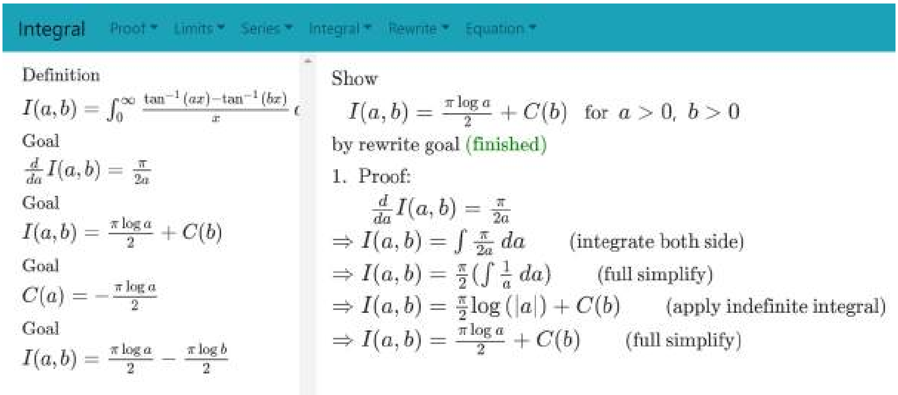
Based on the above rules and proof methods, Iscalc supports performing a variety of symbolic computation, including showing inequalities, checking convergence, evaluating limits, and performing indefinite and definite integrals. It is also possible to build higher-level automation on top of the rules. An implementation of Slagle's method is inherited from [18]. It performs best-first search using algorithmic and heuristic steps for performing an integral. If the search succeeds, it outputs a sequence of rules to apply, which can then be replayed in Iscalc.

The user interface of Iscalc is mostly inherited from [18]. The primary goal is to provide a visual interface that feels similar to that of a computer algebra system, and which allows mostly point-and-click based interactions. In particular, computation steps are performed by selecting rules to apply from the menu. For certain rules, the user may need to select a subexpression of the current expression to apply the rule on, and/or choose from suggestions given by the computer (e.g. when rewriting using identities).

Additional features in the current work, such as book and file hierarchy, and proof methods, are also supported in the user interface. This includes display and navigation of book and file contents. To begin the proof of an equation, the user selects from the menu one of the proof methods in Sect. 2.5. The structured computation is then displayed in a reader-friendly format. An example showing display of file contents and a computation is given in Fig. 2.

3 Examples

We applied Iscalc on computations of limits, indefinite integrals, and definite integrals from a variety of sources. Three sources are inherited from [18]: an exam preparation book (Tongji), online problem lists by D. Kouba [13], and the MIT integration Bee [1]. The range of applicability is greater on these problem sets. For example, we can now perform all examples in the exponentials and trigonometric category from D. Kouba's problem lists, while the previous work



Definition

$I(a,b) = \int_0^\infty \frac{\tan^{-1}(ax)-\tan^{-1}(bx)}{x}$ c

Goal

$\frac{d}{da} I(a,b) = \frac{\pi}{2a}$

Goal

$I(a,b) = \frac{\pi \log a}{2} + C(b)$

Goal

$C(a) = -\frac{\pi \log a}{2}$

Goal

$I(a,b) = \frac{\pi \log a}{2} - \frac{\pi \log b}{2}$

Show

$I(a,b) = \frac{\pi \log a}{2} + C(b)$ for $a > 0,\ b > 0$

by rewrite goal (finished)

1. Proof:

$\frac{d}{da} I(a,b) = \frac{\pi}{2a}$

$\Rightarrow I(a,b) = \int \frac{\pi}{2a}\, da$ (integrate both side)

$\Rightarrow I(a,b) = \frac{\pi}{2}(\int \frac{1}{a}\, da)$ (full simplify)

$\Rightarrow I(a,b) = \frac{\pi}{2}\log(|a|) + C(b)$ (apply indefinite integral)

$\Rightarrow I(a,b) = \frac{\pi \log a}{2} + C(b)$ (full simplify)

Fig. 2. Screenshot of the user interface, showing part of the example given in Sect. 3.1. The menu groups related rules into categories. The *Proof* category contains general actions such as proof by calculation and induction. The remaining five menu categories contain rewriting rules. The left side of the main window shows division of the computation into several parts, and the right side shows the selected part as a series of computation steps. On the bottom (not shown) are space for users to enter additional information for a computation step.

can perform only 7/12 and 22/27 examples respectively, due to limitations of SymPy as well as other unsupported features.

The main additional benchmark comes from the textbook *Inside Interesting Integrals* [17]. 71 integral calculations are performed in Iscalc, covering about half the content of the book, including early results about Gamma and zeta functions. Many of the remaining examples involve complex numbers and contour integration, which are not supported by the current version of the tool.

Next, we illustrate some special functionality of Iscalc using examples. From these examples, we wish to emphasize how different algorithms and rules described in Sect. 2.3 and 2.4 interact with each other, enabling a computation process that is very close to human writing.

3.1 Working with Indefinite Integrals and *C*

The goal is to evaluate Frullani's integral (Sect. 3.3 of [17]).

$$I(a,b) = \int_0^\infty \frac{\tan^{-1}(ax) - \tan^{-1}(bx)}{x}\, dx$$

under the condition $a > 0, b > 0$. The computation starts by computing $\frac{d}{da} I(a,b) = \frac{\pi}{2a}$, which follows by exchanging derivative and integral, then using the formula for the definite integral $\int_0^\infty \frac{1}{u^2+1}\, dx$. The key step is integrating both sides of $\frac{d}{da} I(a,b) = \frac{\pi}{2a}$ using rule IntegralEquation to obtain $I(a,b) = \int \frac{\pi}{2a}\, da$, which evaluates to

$$I(a,b) = \frac{\pi \log a}{2} + C(b)$$

Here it is important to keep track of the dependency of the constant in $\int \frac{\pi}{2a}\,da$ on the variable b, which is kept in the argument *deps* of the expression. This variable is then shown explicitly as an argument to the Skolem term C when the indefinite integral is evaluated.

Next, substitute b by a in the above equation, and from $I(a,a) = 0$ obtain $C(a) = -\frac{\pi \log a}{2}$. Substituting back in the above equation gives the final answer

$$I(a,b) = \frac{\pi \log a}{2} - \frac{\pi \log b}{2}.$$

The entire computation can be carried out in Iscalc much as described above, consisting one definition and four goals, and using 17 rule applications.

3.2 Wellformedness Checks

An example from Sect. 2.3 in [17], illustrating partial fraction decomposition, involves computing the following integral:

$$I(a) = \int_0^\infty \frac{1}{x^4 + 2x^2\cos(2a) + 1}\,dx$$

under the condition $\cos(a) \neq 0$. One particularly tricky point is that it is not obvious why the denominator is always nonzero. This cannot be shown automatically by Iscalc. However, we can state a separate goal showing this fact by case analysis. One of the step during the computation involves an integral with the same denominator, but with bounds $(-\infty, \infty)$, so we perform the check without any assumption on x.

We perform case analysis on whether x is equal to 0. If $x = 0$ then the goal simply reduces to $1 \neq 0$. If $x \neq 0$, we rewrite the goal as follows (the name of the rule applied is shown at right):

$$\begin{aligned}
&x^4 + 2x^2\cos(2a) + 1 \\
&= (x^2-1)^2 + 2x^2(1+\cos(2a)) && \text{(Equation)} \\
&= (x^2-1)^2 + 2x^2(1+(2\cos^2(a)-1)) && \text{(ApplyIdentity)} \\
&= 4x^2\cos^2(a) + (x^2-1)^2 && \text{(FullSimplify)}
\end{aligned}$$

Now, from $x \neq 0$ and $\cos(a) \neq 0$ we get $4x^2\cos^2(a) > 0$. Also $(x^2-1)^2 \geq 0$, so the whole expression is greater than zero (and hence nonzero). The inequality checking algorithm in Sect. 2.3 is able to perform this reasoning automatically, hence showing the expression in the integral is well-defined. Interestingly, the answer $\frac{\pi}{4\cos(a)}$ given in the book is not fully correct. It only holds when $\cos(a) > 0$. If $\cos(a) < 0$ the correct answer is $-\frac{\pi}{4\cos(a)}$ (we can easily check there is a mistake since the integrand is always positive).

3.3 Convergence Checks

For the final example, we illustrate integration using series, as well as checking convergence. The example comes from Sect. 5 of [17]. The goal is to evaluate

$$\int_0^1 \frac{\log(1+x)}{x}\,dx$$

The technique used is to expand the Taylor series for $\log(1+x)$ (using rule SeriesExpansionIdentity), then exchange integration and summation. During the exchange the body of the sum and integral is $\frac{(-1)^n x^n}{n+1}$. As the body changes sign for different values of n, there is potential danger that the sum is not *absolutely convergent*, and the exchange of sum and integral is incorrect even if the final answer is finite. To exclude this possibility, lscalc requires the user to first show the convergence of $\sum_{n=0}^{\infty} \int_0^1 \frac{x^n}{n+1}\,dx$. This is checked after the computation

$$\sum_{n=0}^{\infty} \int_0^1 \frac{x^n}{n+1}\,dx = \sum_{n=0}^{\infty} \frac{1}{n+1} \int_0^1 x^n\,dx = \sum_{n=0}^{\infty} \frac{1}{(n+1)^2}$$

which is convergent by the p-series test implemented within lscalc. This shows the exchange of sum and integral is indeed safe. The final result of the integral is $\frac{\pi}{12}$, which can be computed in lscalc using 10 rule applications (including 3 for showing convergence), assuming the value of some standard infinite series is already known.

4 Discussion

While there has been a long line of research on visual user-interfaces for interactive theorem proving, one persistent issue is that they are mostly limited to simple examples or narrow application areas. For large scale formalizations, the number of actions the user can perform steadily increases, so it becomes more and more difficult to organize them in the user interface. Our work can be seen as an exploration of how far we can go in the limited, but still wide area of symbolic computation. We believe the results are positive. In particular, the following design decisions contribute to controlling complexity:

- Apply rules automatically as much as possible, so they never need to be explicitly selected by the user (e.g. normalization and inequality checking).
- Group related identities into a single rule (e.g. integrals, series expansions, etc.). After the user selects one of these rules, performing matching on the list of available identities and provide choices to the user.
- Group related rules into categories. For example, rules for evaluating integrals, rules for series manipulation, etc. This results in a two-level menu where the user may find appropriate rules more easily.

The end result is that the user does not need to recall names of any existing identity (in fact no names are assigned at all). Instead, all results are either applied automatically, or selected after matching from a list of suggested choices.

4.1 Related Work

There is a large body of work combining theorem proving and symbolic computation, and in user interface design for theorem provers. Some earlier works include Harrison and Théry's "skeptic's" approach to invoking computer algebra systems from a theorem prover [12], and Bauer et al's Analytica [5], which implements automatic theorem proving for elementary analysis within Mathematica. We leave a detailed review to [18,19]. More recently, Lewis and Wu [14] implemented a bi-directional interface between Lean [16] and Mathematica. Donato et al. designed an interface for constructing proofs using drag-and-drop actions [7].

There are also many implementations of proof procedures related to computer algebra. For example, the tool MetiTarski for proving inequalities by Akbarpour and Paulson [2], and the heuristic-based prover Polya by Avigad et al [4]. For computation of limits, Eberl implemented verified computation of asymptotics with generated proofs in Isabelle [8]. We do not claim our procedures to be more effective than the ones listed above, but focus on their combination with user guidance to allow performing more complex symbolic computations.

5 Conclusion

In this paper, we introduced Iscalc for performing symbolic computation interactively, as a significant extension to the system described in [18]. This results in a more extensible tool with greater range of applicability, in particular able to check difficult computations from the textbook [17], and find some mistakes in the process.

In future work, we wish to extend the functionality of Iscalc to handle complex numbers, multiple integrals, and vector calculus. One particularly interesting question is how to support evaluation of contour integrals (the formalization of which have been done in Isabelle by Li and Paulson [15]). On the applications side, we intend to explore verification of control systems [3].

Finally, more work would be required to extend the proof reconstruction in [18] to the larger set of functionality available, as well as linking with library of theorems in analysis. The custom language of expressions defined here is independent of particular choice of logical foundation, hence proof reconstruction should be possible in any interactive theorem prover.

Acknowledgements. This work was partially supported by the National Natural Science Foundation of China under Grant Nos. 62002351 and 62032024. Part of this work is presented at the IPAM workshop *Machine Assisted Proofs*. We thank audience members at the talk for valuable feedback that helped to improve this work.

References

1. MIT Integration Bee. https://www.mit.edu/~pax/integrationbee.html. Accessed 22 Jan 2020
2. Akbarpour, B., Paulson, L.C.: Metitarski: an automatic theorem prover for real-valued special functions. J. Autom. Reason. **44**(3), 175–205 (2010). https://doi.org/10.1007/s10817-009-9149-2
3. Åström, K.J., Murray, R.M.: Feedback systems: An introduction for scientists and engineers (2008)
4. Avigad, J., Lewis, R.Y., Roux, C.: A heuristic prover for real inequalities. J. Autom. Reason. **56**(3), 367–386 (2016). https://doi.org/10.1007/s10817-015-9356-y
5. Bauer, A., Clarke, E.M., Zhao, X.: Analytica - an experiment in combining theorem proving and symbolic computation. J. Autom. Reason. **21**(3), 295–325 (1998). https://doi.org/10.1023/A:1006079212546
6. Boldo, S., Lelay, C., Melquiond, G.: Formalization of real analysis: a survey of proof assistants and libraries. Math. Struct. Comput. Sci. **26**(7), 1196–1233 (2016)
7. Donato, P., Strub, P., Werner, B.: A drag-and-drop proof tactic. In: Popescu, A., Zdancewic, S. (eds.) 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2022, Philadelphia, PA, USA, 17–18 January 2022, pp. 197–209. ACM (2022). https://doi.org/10.1145/3497775.3503692
8. Eberl, M.: Verified real asymptotics in Isabelle/HOL. In: Davenport, J.H., Wang, D., Kauers, M., Bradford, R.J. (eds.) Proceedings of the 2019 on International Symposium on Symbolic and Algebraic Computation, ISSAC 2019, Beijing, China, 15–18 July 2019, pp. 147–154. ACM (2019). https://doi.org/10.1145/3326229.3326240
9. Fu, H., Zhong, X., Zeng, Z.: Automated and readable simplification of trigonometric expressions. Math. Comput. Model. **44**(11–12), 1169–1177 (2006). https://doi.org/10.1016/j.mcm.2006.04.002
10. Gruntz, D.: On computing limits in a symbolic manipulation system (1996)
11. Guardeño, A.J.D., Riera, M.P., Malumbres, J.L.V.: The misfortunes of a trio of mathematicians using computer algebra systems. Can we trust in them? Not. Am. Math. Soc. **61**, 1249–1252 (2014)
12. Harrison, J., Théry, L.: A skeptic's approach to combining HOL and Maple. J. Autom. Reason. **21**(3), 279–294 (1998). https://doi.org/10.1023/A:1006023127567
13. Kouba, D.A.: The calculus page problems list. https://www.math.ucdavis.edu/kouba/ProblemsList.html. Accessed 22 Jan 2020
14. Lewis, R.Y., Wu, M.: A bi-directional extensible interface between lean and mathematica. J. Autom. Reason. **66**(2), 215–238 (2022). https://doi.org/10.1007/s10817-021-09611-1
15. Li, W., Paulson, L.C.: Evaluating winding numbers and counting complex roots through Cauchy indices in Isabelle/HOL. J. Autom. Reason. **64**(2), 331–360 (2020). https://doi.org/10.1007/s10817-019-09521-3
16. Moura, L., Ullrich, S.: The lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 625–635. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_37
17. Nahin, P.J.: Inside interesting integrals. Undergraduate Lecture Notes in Physics (2014)

18. Xu, R., Li, L., Zhan, B.: Verified interactive computation of definite integrals. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 485–503. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_28
19. Zhan, B., Ji, Z., Zhou, W., Xiang, C., Hou, J., Sun, W.: Design of point-and-click user interfaces for proof assistants. In: Ait-Ameur, Y., Qin, S. (eds.) ICFEM 2019. LNCS, vol. 11852, pp. 86–103. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32409-4_6

Author Index

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 591–592, 2023.
https://doi.org/10.1007/978-3-031-38499-8