Amal Ahmed (Ed.)

ARCoSS

LNCS 10801

Programming Languages and Systems

27th European Symposium on Programming, ESOP 2018
Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018
Thessaloniki, Greece, April 14–20, 2018, Proceedings

ETAPS
EUROPEAN JOINT CONFERENCES ON THEORY & PRACTICE OF SOFTWARE

Springer Open

EXTRAS ONLINE

Amal Ahmed (Ed.)

Programming Languages and Systems

27th European Symposium on Programming, ESOP 2018
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2018
Thessaloniki, Greece, April 14–20, 2018
Proceedings

Springer Open

Editor
Amal Ahmed
Northeastern University
Boston, MA
USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-89883-4 ISBN 978-3-319-89884-1 (eBook)
https://doi.org/10.1007/978-3-319-89884-1

Library of Congress Control Number: 2018940640

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

Printed on acid-free paper

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

ETAPS Foreword

Welcome to the proceedings of ETAPS 2018! After a somewhat coldish ETAPS 2017 in Uppsala in the north, ETAPS this year took place in Thessaloniki, Greece. I am happy to announce that this is the first ETAPS with gold open access proceedings. This means that all papers are accessible by anyone for free.

ETAPS 2018 was the 21st instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference established in 1998, and consists of five conferences: ESOP, FASE, FoSSaCS, TACAS, and POST. Each conference has its own Program Committee (PC) and its own Steering Committee. The conferences cover various aspects of software systems, ranging from theoretical computer science to foundations to programming language developments, analysis tools, formal approaches to software engineering, and security. Organizing these conferences in a coherent, highly synchronized conference program facilitates participation in an exciting event, offering attendees the possibility to meet many researchers working in different directions in the field, and to easily attend talks of different conferences. Before and after the main conference, numerous satellite workshops take place and attract many researchers from all over the globe.

ETAPS 2018 received 479 submissions in total, 144 of which were accepted, yielding an overall acceptance rate of 30%. I thank all the authors for their interest in ETAPS, all the reviewers for their peer reviewing efforts, the PC members for their contributions, and in particular the PC (co-)chairs for their hard work in running this entire intensive process. Last but not least, my congratulations to all authors of the accepted papers!

ETAPS 2018 was enriched by the unifying invited speaker Martin Abadi (Google Brain, USA) and the conference-specific invited speakers (FASE) Pamela Zave (AT & T Labs, USA), (POST) Benjamin C. Pierce (University of Pennsylvania, USA), and (ESOP) Derek Dreyer (Max Planck Institute for Software Systems, Germany). Invited tutorials were provided by Armin Biere (Johannes Kepler University, Linz, Austria) on modern SAT solving and Fabio Somenzi (University of Colorado, Boulder, USA) on hardware verification. My sincere thanks to all these speakers for their inspiring and interesting talks!

ETAPS 2018 took place in Thessaloniki, Greece, and was organised by the Department of Informatics of the Aristotle University of Thessaloniki. The university was founded in 1925 and currently has around 75,000 students; it is the largest university in Greece. ETAPS 2018 was further supported by the following associations and societies: ETAPS e.V., EATCS (European Association for Theoretical Computer Science), EAPLS (European Association for Programming Languages and Systems), and EASST (European Association of Software Science and Technology). The local organization team consisted of Panagiotis Katsaros (general chair), Ioannis Stamelos, Lefteris Angelis, George Rahonis, Nick Bassiliades, Alexander Chatzigeorgiou, Ezio Bartocci, Simon Bliudze, Emmanouela Stachtiari, Kyriakos Georgiadis, and Petros Stratis (EasyConferences).

The overall planning for ETAPS is the main responsibility of the Steering Committee, and in particular of its Executive Board. The ETAPS Steering Committee consists of an Executive Board and representatives of the individual ETAPS conferences, as well as representatives of EATCS, EAPLS, and EASST. The Executive Board consists of Gilles Barthe (Madrid), Holger Hermanns (Saarbrücken), Joost-Pieter Katoen (chair, Aachen and Twente), Gerald Lüttgen (Bamberg), Vladimiro Sassone (Southampton), Tarmo Uustalu (Tallinn), and Lenore Zuck (Chicago). Other members of the Steering Committee are: Wil van der Aalst (Aachen), Parosh Abdulla (Uppsala), Amal Ahmed (Boston), Christel Baier (Dresden), Lujo Bauer (Pittsburgh), Dirk Beyer (Munich), Mikolaj Bojanczyk (Warsaw), Luis Caires (Lisbon), Jurriaan Hage (Utrecht), Rainer Hähnle (Darmstadt), Reiko Heckel (Leicester), Marieke Huisman (Twente), Panagiotis Katsaros (Thessaloniki), Ralf Küsters (Stuttgart), Ugo Dal Lago (Bologna), Kim G. Larsen (Aalborg), Matteo Maffei (Vienna), Tiziana Margaria (Limerick), Flemming Nielson (Copenhagen), Catuscia Palamidessi (Palaiseau), Andrew M. Pitts (Cambridge), Alessandra Russo (London), Dave Sands (Göteborg), Don Sannella (Edinburgh), Andy Schürr (Darmstadt), Alex Simpson (Ljubljana), Gabriele Taentzer (Marburg), Peter Thiemann (Freiburg), Jan Vitek (Prague), Tomas Vojnar (Brno), and Lijun Zhang (Beijing).

I would like to take this opportunity to thank all speakers, attendees, organizers of the satellite workshops, and Springer for their support. I hope you all enjoy the proceedings of ETAPS 2018. Finally, a big thanks to Panagiotis and his local organization team for all their enormous efforts that led to a fantastic ETAPS in Thessaloniki!

February 2018 Joost-Pieter Katoen

Preface

This volume contains the papers presented at the 27th European Symposium on Programming (ESOP 2018) held April 16–19, 2018, in Thessaloniki, Greece. ESOP is one of the European Joint Conferences on Theory and Practice of Software (ETAPS). It is devoted to fundamental issues in the specification, design, analysis, and implementation of programming languages and systems.

The 36 papers in this volume were selected from 114 submissions based on originality and quality. Each submission was reviewed by three to six Program Committee (PC) members and external reviewers, with an average of 3.3 reviews per paper. Authors were given a chance to respond to these reviews during the rebuttal period from December 6 to 8, 2017. All submissions, reviews, and author responses were considered during the online discussion, which identified 74 submissions to be discussed further at the physical PC meeting held at Inria Paris, December 13–14, 2017. Each paper was assigned a guardian, who was responsible for making sure that external reviews were solicited if there was not enough non-conflicted expertise among the PC, and for presenting a summary of the reviews and author responses at the PC meeting. All non-conflicted PC members participated in the discussion of a paper's merits. PC members wrote reactions to author responses, including summaries of online discussions and discussions during the physical PC meeting, so as to help the authors understand decisions. Papers co-authored by members of the PC were held to a higher standard and discussed toward the end of the physical PC meeting. There were ten such submissions and five were accepted. Papers for which the program chair had a conflict of interest were kindly handled by Fritz Henglein.

My sincere thanks to all who contributed to the success of the conference. This includes the authors who submitted papers for consideration; the external reviewers, who provided timely expert reviews, sometimes on short notice; and the PC, who worked hard to provide extensive reviews, engaged in high-quality discussions about the submissions, and added detailed comments to help authors understand the PC discussion and decisions. I am grateful to the past ESOP PC chairs, particularly Jan Vitek and Hongseok Yang, and to the ESOP SC chairs, Giuseppe Castagna and Peter Thiemann, who helped with numerous procedural matters. I would like to thank the ETAPS SC chair, Joost-Pieter Katoen, for his amazing work and his responsiveness. HotCRP was used to handle submissions and online discussion, and helped smoothly run the physical PC meeting. Finally, I would like to thank Cătălin Hriţcu for sponsoring the physical PC meeting through ERC grant SECOMP, Mathieu Mourey and the Inria Paris staff for their help organizing the meeting, and William Bowman for assisting with the PC meeting.

February 2018 Amal Ahmed

Organization

Program Committee

Amal Ahmed	Northeastern University, USA and Inria, France
Nick Benton	Facebook, UK
Josh Berdine	Facebook, UK
Viviana Bono	Università di Torino, Italy
Dominique Devriese	KU Leuven, Belgium
Marco Gaboardi	University at Buffalo, SUNY, USA
Roberto Giacobazzi	Università di Verona, Italy and IMDEA Software Institute, Spain
Philipp Haller	KTH Royal Institute of Technology, Sweden
Matthew Hammer	University of Colorado Boulder, USA
Fritz Henglein	University of Copenhagen, Denmark
Jan Hoffmann	Carnegie Mellon University, USA
Cătălin Hriţcu	Inria Paris, France
Suresh Jagannathan	Purdue University, USA
Limin Jia	Carnegie Mellon University, USA
Naoki Kobayashi	University of Tokyo, Japan
Xavier Leroy	Inria Paris, France
Aleksandar Nanevski	IMDEA Software Institute, Spain
Michael Norrish	Data61 and CSIRO, Australia
Andreas Rossberg	Google, Germany
Davide Sangiorgi	Università di Bologna, Italy and Inria, France
Peter Sewell	University of Cambridge, UK
Éric Tanter	University of Chile, Chile
Niki Vazou	University of Maryland, USA
Steve Zdancewic	University of Pennsylvania, USA

Additional Reviewers

Danel Ahman
S. Akshay
Aws Albarghouthi
Jade Alglave
Vincenzo Arceri
Samik Basu
Gavin Bierman
Filippo Bonchi
Thierry Coquand
Mariangiola Dezani
Derek Dreyer
Ronald Garcia
Deepak Garg
Samir Genaim
Victor Gomes
Peter Habermehl
Matthew Hague
Justin Hsu

Zhenjiang Hu
Peter Jipsen
Shin-ya Katsumata
Andrew Kennedy
Heidy Khlaaf
Neelakantan Krishnaswami
César Kunz
Ugo Dal Lago
Paul Levy
Kenji Maillard
Roman Manevich
Paulo Mateus
Antoine Miné
Stefan Monnier
Andrzej Murawski
Anders Møller
Vivek Notani
Andreas Nuyts
Paulo Oliva
Dominic Orchard
Luca Padovani
Brigitte Pientka
Benjamin C. Pierce
Andreas Podelski
Chris Poskitt
Francesco Ranzato
Andrey Rybalchenko
Sriram Sankaranarayanan
Tetsuya Sato
Sandro Stucki
Zachary Tatlock
Bernardo Toninho
Viktor Vafeiadis

RustBelt: Logical Foundations for the Future of Safe Systems Programming

Derek Dreyer

Max Planck Institute for Software Systems (MPI-SWS), Germany
dreyer@mpi-sws.org

Abstract. Rust is a new systems programming language, developed at Mozilla, that promises to overcome the seemingly fundamental tradeoff in language design between high-level safety guarantees and low-level control over resource management. Unfortunately, none of Rust's safety claims have been formally proven, and there is good reason to question whether they actually hold. Specifically, Rust employs a strong, ownership-based type system, but then extends the expressive power of this core type system through libraries that internally use unsafe features.

In this talk, I will present RustBelt (http://plv.mpi-sws.org/rustbelt), the first formal (and machine-checked) safety proof for a language representing a realistic subset of Rust. Our proof is extensible in the sense that, for each new Rust library that uses unsafe features, we can say what verification condition it must satisfy in order for it to be deemed a safe extension to the language. We have carried out this verification for some of the most important libraries that are used throughout the Rust ecosystem.

After reviewing some essential features of the Rust language, I will describe the high-level structure of the RustBelt verification and then delve into detail about the secret weapon that makes RustBelt possible: the Iris framework for higher-order concurrent separation logic in Coq (http://iris-project.org). I will explain by example how Iris generalizes the expressive power of O'Hearn's original concurrent separation logic in ways that are essential for verifying the safety of Rust libraries. I will not assume any prior familiarity with concurrent separation logic or Rust.

This is joint work with Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and the rest of the Iris team.

Contents

Language Design

Consistent Subtyping for All

Ningning Xie(✉), Xuan Bi, and Bruno C. d. S. Oliveira

The University of Hong Kong, Pokfulam, Hong Kong
{nnxie,xbi,bruno}@cs.hku.hk

Abstract. Consistent subtyping is employed in some gradual type systems to validate type conversions. The original definition by Siek and Taha serves as a guideline for designing gradual type systems with subtyping. Polymorphic types à la System F also induce a subtyping relation that relates polymorphic types to their instantiations. However Siek and Taha's definition is not adequate for polymorphic subtyping. The first goal of this paper is to propose a generalization of consistent subtyping that is adequate for polymorphic subtyping, and subsumes the original definition by Siek and Taha. The new definition of consistent subtyping provides novel insights with respect to previous polymorphic gradual type systems, which did not employ consistent subtyping. The second goal of this paper is to present a gradually typed calculus for implicit (higher-rank) polymorphism that uses our new notion of consistent subtyping. We develop both declarative and (bidirectional) algorithmic versions for the type system. We prove that the new calculus satisfies all static aspects of the refined criteria for gradual typing, which are mechanically formalized using the Coq proof assistant.

1 Introduction

Gradual typing [21] is an increasingly popular topic in both programming language practice and theory. On the practical side there is a growing number of programming languages adopting gradual typing. Those languages include Clojure [6], Python [27], TypeScript [5], Hack [26], and the addition of Dynamic to C# [4], to cite a few. On the theoretical side, recent years have seen a large body of research that defines the foundations of gradual typing [8,9,13], explores their use for both functional and object-oriented programming [21,22], as well as its applications to many other areas [3,24].

A key concept in gradual type systems is *consistency* [21]. Consistency weakens type equality to allow for the presence of *unknown* types. In some gradual type systems with subtyping, consistency is combined with subtyping to give rise to the notion of *consistent subtyping* [22]. Consistent subtyping is employed by gradual type systems to validate type conversions arising from conventional subtyping. One nice feature of consistent subtyping is that it is derivable from the more primitive notions of *consistency* and *subtyping*. As Siek and Taha [22] put it this shows that "*gradual typing and subtyping are orthogonal and can be combined in a principled fashion*". Thus consistent subtyping is often used as a guideline for designing gradual type systems with subtyping.

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 3–30, 2018.
https://doi.org/10.1007/978-3-319-89884-1_1

Unfortunately, as noted by Garcia et al. [13], notions of consistency and/or consistent subtyping "*become more difficult to adapt as type systems get more complex*". In particular, for the case of type systems with subtyping, certain kinds of subtyping do not fit well with the original definition of consistent subtyping by Siek and Taha [22]. One important case where such mismatch happens is in type systems supporting implicit (higher-rank) polymorphism [11,18]. It is well-known that polymorphic types à la System F induce a subtyping relation that relates polymorphic types to their instantiations [16,17]. However Siek and Taha's [22] definition is not adequate for this kind of subtyping. Moreover the current framework for *Abstracting Gradual Typing* (AGT) [13] also does not account for polymorphism, with the authors acknowledging that this is one of the interesting avenues for future work.

Existing work on gradual type systems with polymorphism does not use consistent subtyping. The Polymorphic Blame Calculus ($\lambda\mathsf{B}$) [1] is an *explicitly* polymorphic calculus with explicit casts, which is often used as a target language for gradual type systems with polymorphism. In $\lambda\mathsf{B}$ a notion of *compatibility* is employed to validate conversions allowed by casts. Interestingly $\lambda\mathsf{B}$ *allows conversions from polymorphic types to their instantiations.* For example, it is possible to cast a value with type $\forall a.a \to a$ into $\mathsf{Int} \to \mathsf{Int}$. Thus an important remark here is that while $\lambda\mathsf{B}$ is explicitly polymorphic, casting and conversions are closer to *implicit* polymorphism. That is, in a conventional explicitly polymorphic calculus (such as System F), the primary notion is type equality, where instantiation is not taken into account. Thus the types $\forall a.a \to a$ and $\mathsf{Int} \to \mathsf{Int}$ are deemed *incompatible.* However in *implicitly* polymorphic calculi [11,18] $\forall a.a \to a$ and $\mathsf{Int} \to \mathsf{Int}$ are deemed *compatible*, since the latter type is an instantiation of the former. Therefore $\lambda\mathsf{B}$ is in a sense a hybrid between implicit and explicit polymorphism, utilizing type equality (à la System F) for validating applications, and *compatibility* for validating casts.

An alternative approach to polymorphism has recently been proposed by Igarashi et al. [14]. Like $\lambda\mathsf{B}$ their calculus is explicitly polymorphic. However, in that work they employ type consistency to validate cast conversions, and forbid conversions from $\forall a.a \to a$ to $\mathsf{Int} \to \mathsf{Int}$. This makes their casts closer to explicit polymorphism, in contrast to $\lambda\mathsf{B}$. Nonetheless, there is still same flavour of implicit polymorphism in their calculus when it comes to interactions between dynamically typed and polymorphically typed code. For example, in their calculus type consistency allows types such as $\forall a.a \to \mathsf{Int}$ to be related to $\star \to \mathsf{Int}$, where some sort of (implicit) polymorphic subtyping is involved.

The first goal of this paper is to study the gradually typed subtyping and consistent subtyping relations for *predicative implicit polymorphism.* To accomplish this, we first show how to reconcile consistent subtyping with polymorphism by generalizing the original consistent subtyping definition by Siek and Taha [22]. The new definition of consistent subtyping can deal with polymorphism, and preserves the orthogonality between consistency and subtyping. To slightly rephrase Siek and Taha [22], the motto of our paper is that:

> *Gradual typing and* **polymorphism** *are orthogonal and can be combined in a principled fashion.*[1]

With the insights gained from our work, we argue that, for implicit polymorphism, Ahmed et al.'s [1] notion of compatibility is too permissive (i.e. too many programs are allowed to type-check), and that Igarashi et al.'s [14] notion of type consistency is too conservative. As a step towards an algorithmic version of consistent subtyping, we present a syntax-directed version of consistent subtyping that is sound and complete with respect to our formal definition of consistent subtyping. The syntax-directed version of consistent subtyping is remarkably simple and well-behaved, without the ad-hoc *restriction* operator [22]. Moreover, to further illustrate the generality of our consistent subtyping definition, we show that it can also account for *top types*, which cannot be dealt with by Siek and Taha's [22] definition either.

The second goal of this paper is to present a (source-level) gradually typed calculus for (predicative) implicit higher-rank polymorphism that uses our new notion of consistent subtyping. As far as we are aware, there is no work on bridging the gap between implicit higher-rank polymorphism and gradual typing, which is interesting for two reasons. On one hand, modern functional languages (such as Haskell) employ sophisticated type-inference algorithms that, aided by type annotations, can deal with implicit higher-rank polymorphism. So a natural question is how gradual typing can be integrated in such languages. On the other hand, there is several existing work on integrating *explicit* polymorphism into gradual typing [1,14]. Yet no work investigates how to move such expressive power into a source language with implicit polymorphism. Therefore as a step towards gradualizing such type systems, this paper develops both declarative and algorithmic versions for a gradual type system with implicit higher-rank polymorphism. The new calculus brings the expressive power of full implicit higher-rank polymorphic into a gradually typed source language. We prove that our calculus satisfies all of the *static* aspects of the refined criteria for gradual typing [25], while discussing some issues related with the *dynamic guarantee.*

In summary, the contributions of this paper are:

- We define a framework for consistent subtyping with:
  - a new definition of consistent subtyping that subsumes and generalizes that of Siek and Taha [22], and can deal with polymorphism and top types.
  - a syntax-directed version of consistent subtyping that is sound and complete with respect to our definition of consistent subtyping, but still guesses polymorphic instantiations.

[1] Note here that we borrow Siek and Taha's [22] motto mostly to talk about the static semantics. As Ahmed et al. [1] show there are several non-trivial interactions between polymorphism and casts at the level of the dynamic semantics.

$$\boxed{A <: B}$$

$$\mathsf{Int} <: \mathsf{Int} \qquad \mathsf{Bool} <: \mathsf{Bool} \qquad \mathsf{Float} <: \mathsf{Float} \qquad \mathsf{Int} <: \mathsf{Float}$$

$$\frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2} \qquad [l_i : A_i^{i \in 1 \ldots n+m}] <: [l_i : A_i^{i \in 1 \ldots n}] \qquad \star <: \star$$

$$\boxed{A \sim B}$$

$$A \sim A \qquad A \sim \star \qquad \star \sim A \qquad \frac{A_1 \sim B_1 \qquad A_2 \sim B_2}{A_1 \to A_2 \sim B_1 \to B_2} \qquad \frac{A_i \sim B_i}{[l_i : A_i] \sim [l_i : B_i]}$$

Fig. 1. Subtyping and type consistency in $\mathbf{FOb}^{?}_{<:}$

- Based on consistent subtyping, we present a declarative gradual type system with predicative implicit higher-rank polymorphism. We prove that our calculus satisfies the static aspects of the refined criteria for gradual typing [25], and is type-safe by a type-directed translation to $\lambda\mathsf{B}$, and thus hereditarily preserves parametricity [2].
- We present a complete and sound bidirectional algorithm for implementing the declarative system based on the design principle of Garcia and Cimini [12] and the approach of Dunfield and Krishnaswami [11].
- All of the metatheory of this paper, except some manual proofs for the algorithmic type system, has been mechanically formalized in Coq[2].

2 Background and Motivation

In this section we review a simple gradually typed language with objects [22], to introduce the concept of consistency subtyping. We also briefly talk about the Odersky-Läufer type system for higher-rank types [17], which serves as the original language on which our gradually typed calculus with implicit higher-rank polymorphism is based.

2.1 Gradual Subtyping

Siek and Taha [22] developed a gradual typed system for object-oriented languages that they call $\mathbf{FOb}^{?}_{<:}$. Central to gradual typing is the concept of *consistency* (written $\sim$) between gradual types, which are types that may involve the unknown type $\star$. The intuition is that consistency relaxes the structure of a type system to tolerate unknown positions in a gradual type. They also defined the subtyping relation in a way that static type safety is preserved. Their key

[2] All supplementary materials are available at https://bitbucket.org/xieningning/consistent-subtyping.

insight is that the unknown type $\star$ is neutral to subtyping, with only $\star <: \star$. Both relations are found in Fig. 1.

A primary contribution of their work is to show that consistency and subtyping are orthogonal. To compose subtyping and consistency, Siek and Taha [22] defined *consistent subtyping* (written $\lesssim$) in two equivalent ways:

Definition 1 (Consistent Subtyping à la Siek and Taha [22])

- *$A \lesssim B$ if and only if $A \sim C$ and $C <: B$ for some C.*
- *$A \lesssim B$ if and only if $A <: C$ and $C \sim B$ for some C.*

Both definitions are non-deterministic because of the intermediate type C. To remove non-determinism, they proposed a so-called *restriction operator*, written $A|_B$ that masks off the parts of a type A that are unknown in a type B.

$$\begin{aligned}
A|_B = \textbf{case}\ \ & A, B\ \ \textbf{of}\ \mid (-, \star) \Rightarrow \star \\
&\mid A_1 \rightarrow A_2, B_1 \rightarrow B_2 = A_1|_{B_1} \rightarrow A_2|_{B_2} \\
&\mid [l_1 : A_1, ..., l_n : A_n], [l_1 : B_1, ..., l_m : B_m]\ \ \textbf{if}\ \ n \leq m \Rightarrow [l_1 : A_1|_{B_1}, ..., l_n : A_n|_{B_n}] \\
&\mid [l_1 : A_1, ..., l_n : A_n], [l_1 : B_1, ..., l_m : B_m]\ \ \textbf{if}\ \ n > m \Rightarrow \\
&\qquad [l_1 : A_1|_{B_1}, ..., l_m : A_m|_{B_m}, ..., l_n : A_n] \\
&\mid \textbf{otherwise} \Rightarrow A
\end{aligned}$$

With the restriction operator, consistent subtyping is simply defined as $A \lesssim B \equiv A|_B <: B|_A$. Then they proved that this definition is equivalent to Definition 1.

2.2 The Odersky-Läufer Type System

The calculus we are combining gradual typing with is the well-established predicative type system for higher-rank types proposed by Odersky and Läufer [17]. One difference is that, for simplicity, we do not account for a let expression, as there is already existing work about gradual type systems with let expressions and let generalization (for example, see Garcia and Cimini [12]). Similar techniques can be applied to our calculus to enable let generalization.

The syntax of the type system, along with the typing and subtyping judgments is given in Fig. 2. An implicit assumption throughout the paper is that variables in contexts are distinct. We save the explanations for the static semantics to Sect. 4, where we present our gradually typed version of the calculus.

2.3 Motivation: Gradually Typed Higher-Rank Polymorphism

Our work combines implicit (higher-rank) polymorphism with gradual typing. As is well known, a gradually typed language supports both fully static and fully dynamic checking of program properties, as well as the continuum between these two extremes. It also offers programmers fine-grained control over the static-to-dynamic spectrum, i.e., a program can be evolved by introducing more or less precise types as needed [13].

Expressions	$e ::= x \mid n \mid \lambda x : A.\ e \mid \lambda x.\ e \mid e\ e$
Types	$A, B ::= \mathsf{Int} \mid a \mid A \to B \mid \forall a.A$
Monotypes	$\tau, \sigma ::= \mathsf{Int} \mid a \mid \tau \to \sigma$
Contexts	$\Psi ::= \varnothing \mid \Psi, x : A \mid \Psi, a$

$$\boxed{\Psi \vdash^{OL} e : A}$$

$$\frac{x : A \in \Psi}{\Psi \vdash^{OL} x : A}\ \textsc{Var} \qquad \frac{}{\Psi \vdash^{OL} n : \mathsf{Int}}\ \textsc{Nat} \qquad \frac{\Psi, x : A \vdash^{OL} e : B}{\Psi \vdash^{OL} \lambda x : A.\ e : A \to B}\ \textsc{LamAnn}$$

$$\frac{\Psi \vdash^{OL} e_1 : A_1 \to A_2 \qquad \Psi \vdash^{OL} e_2 : A_1}{\Psi \vdash^{OL} e_1\ e_2 : A_2}\ \textsc{App} \qquad \frac{\Psi \vdash^{OL} e : A_1 \qquad \Psi \vdash A_1 <: A_2}{\Psi \vdash^{OL} e : A_2}\ \textsc{Sub}$$

$$\frac{\Psi, x : \tau \vdash^{OL} e : B}{\Psi \vdash^{OL} \lambda x.\ e : \tau \to B}\ \textsc{Lam} \qquad \frac{\Psi, a \vdash^{OL} e : A}{\Psi \vdash^{OL} e : \forall a.A}\ \textsc{Gen}$$

$$\boxed{\Psi \vdash A <: B}$$

$$\frac{a \in \Psi}{\Psi \vdash a <: a}\ \text{CS-TVar} \qquad \frac{}{\Psi \vdash \mathsf{Int} <: \mathsf{Int}}\ \text{CS-Int} \qquad \frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] <: B}{\Psi \vdash \forall a.A <: B}\ \textsc{ForallL}$$

$$\frac{\Psi, a \vdash A <: B}{\Psi \vdash A <: \forall a.B}\ \textsc{ForallR} \qquad \frac{\Psi \vdash B_1 <: A_1 \qquad \Psi \vdash A_2 <: B_2}{\Psi \vdash A_1 \to A_2 <: B_1 \to B_2}\ \text{CS-Fun}$$

Fig. 2. Syntax and static semantics of the Odersky-Läufer type system.

Haskell is a language that supports implicit higher-rank polymorphism, but no gradual typing. Therefore some programs that are safe at run-time may be rejected due to the conservativity of the type system. For example, consider the following Haskell program adapted from Jones et al. [18]:

foo :: ([**Int**], [**Char**])
foo = **let** *f x* = (*x* [1, 2] , *x* [′*a*′, ′*b*′]) **in** *f* **reverse**

This program is rejected by Haskell's type checker because Haskell implements the Damas-Milner rule that a lambda-bound argument (such as x) can only have a monotype, i.e., the type checker can only assign x the type $[\mathbf{Int}] \to [\mathbf{Int}]$, or $[\mathbf{Char}] \to [\mathbf{Char}]$, but not $\forall a.[a] \to [a]$. Finding such manual polymorphic annotations can be non-trivial. Instead of rejecting the program outright, due to missing type annotations, gradual typing provides a simple alternative by giving x the unknown type (denoted $\star$). With such typing the same program type-checks and produces $([2, 1], ['b', 'a'])$. By running the program, programmers can gain some additional insight about the run-time behaviour. Then, with such insight, they can also give x a more precise type ($\forall a.[a] \to [a]$) a posteriori so that the program continues to type-check via implicit polymorphism and also grants

Types	$A, B ::= \mathsf{Int} \mid a \mid A \to B \mid \forall a.A \mid \star$
Monotypes	$\tau, \sigma ::= \mathsf{Int} \mid a \mid \tau \to \sigma$
Contexts	$\Psi ::= \varnothing \mid \Psi, x : A \mid \Psi, a$

$$\boxed{A \sim B}$$

$$A \sim A \qquad A \sim \star \qquad \star \sim A \qquad \frac{A_1 \sim B_1 \qquad A_2 \sim B_2}{A_1 \to A_2 \sim B_1 \to B_2} \qquad \frac{A \sim B}{\forall a.A \sim \forall a.B}$$

$$\boxed{\Psi \vdash A <: B}$$

$$\frac{\Psi, a \vdash A <: B}{\Psi \vdash A <: \forall a.B}\ \textsc{S-ForallR} \qquad \frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] <: B}{\Psi \vdash \forall a.A <: B}\ \textsc{S-ForallL} \qquad \frac{a \in \Psi}{\Psi \vdash a <: a}\ \textsc{S-TVar}$$

$$\frac{}{\Psi \vdash \mathsf{Int} <: \mathsf{Int}}\ \textsc{S-Int} \qquad \frac{\Psi \vdash B_1 <: A_1 \qquad \Psi \vdash A_2 <: B_2}{\Psi \vdash A_1 \to A_2 <: B_1 \to B_2}\ \textsc{S-Fun} \qquad \frac{}{\Psi \vdash \star <: \star}\ \textsc{S-Unknown}$$

Fig. 3. Syntax of types, consistency, and subtyping in the declarative system.

more static safety. In this paper, we envision such a language that combines the benefits of both implicit higher-rank polymorphism and gradual typing.

3 Revisiting Consistent Subtyping

In this section we explore the design space of consistent subtyping. We start with the definitions of consistency and subtyping for polymorphic types, and compare with some relevant work. We then discuss the design decisions involved towards our new definition of consistent subtyping, and justify the new definition by demonstrating its equivalence with that of Siek and Taha [22] and the AGT approach [13] on simple types.

The syntax of types is given at the top of Fig. 3. We write A, B for types. Types are either the integer type Int, type variables a, functions types $A \to B$, universal quantification $\forall a.A$, or the unknown type $\star$. Though we only have one base type Int, we also use Bool for the purpose of illustration. Note that monotypes τ contain all types other than the universal quantifier and the unknown type $\star$. We will discuss this restriction when we present the subtyping rules. Contexts Ψ are *ordered* lists of type variable declarations and term variables.

3.1 Consistency and Subtyping

We start by giving the definitions of consistency and subtyping for polymorphic types, and comparing our definitions with the compatibility relation by Ahmed et al. [1] and type consistency by Igarashi et al. [14].

Consistency. The key observation here is that consistency is mostly a structural relation, except that the unknown type $\star$ can be regarded as any type. Following this observation, we naturally extend the definition from Fig. 1 with polymorphic types, as shown at the middle of Fig. 3. In particular a polymorphic type $\forall a.A$ is consistent with another polymorphic type $\forall a.B$ if A is consistent with B.

Subtyping. We express the fact that one type is a polymorphic generalization of another by means of the subtyping judgment $\Psi \vdash A <: B$. Compared with the subtyping rules of Odersky and Läufer [17] in Fig. 2, the only addition is the neutral subtyping of $\star$. Notice that, in the rule S-ForallL, the universal quantifier is only allowed to be instantiated with a *monotype*. The judgment $\Psi \vdash \tau$ checks all the type variables in τ are bound in the context Ψ. For space reasons, we omit the definition. According to the syntax in Fig. 3, monotypes do not include the unknown type $\star$. This is because if we were to allow the unknown type to be used for instantiation, we could have $\forall a.a \to a <: \star \to \star$ by instantiating a with $\star$. Since $\star \to \star$ is consistent with any functions $A \to B$, for instance, $\mathsf{Int} \to \mathsf{Bool}$, this means that we could provide an expression of type $\forall a.a \to a$ to a function where the input type is supposed to be $\mathsf{Int} \to \mathsf{Bool}$. However, as we might expect, $\forall a.a \to a$ is definitely not compatible with $\mathsf{Int} \to \mathsf{Bool}$. This does not hold in any polymorphic type systems without gradual typing. So the gradual type system should not accept it either. (This is the so-called *conservative extension* property that will be made precise in Sect. 4.3.)

Importantly there is a subtle but crucial distinction between a type variable and the unknown type, although they all represent a kind of "arbitrary" type. The unknown type stands for the absence of type information: it could be *any type* at *any instance*. Therefore, the unknown type is consistent with any type, and additional type-checks have to be performed at runtime. On the other hand, a type variable indicates *parametricity*. In other words, a type variable can only be instantiated to a single type. For example, in the type $\forall a.a \to a$, the two occurrences of a represent an arbitrary but single type (e.g., $\mathsf{Int} \to \mathsf{Int}$, $\mathsf{Bool} \to \mathsf{Bool}$), while $\star \to \star$ could be an arbitrary function (e.g., $\mathsf{Int} \to \mathsf{Bool}$) at runtime.

Comparison with Other Relations. In other polymorphic gradual calculi, consistency and subtyping are often mixed up to some extent. In $\lambda\mathsf{B}$ [1], the compatibility relation for polymorphic types is defined as follows:

$$\frac{A \prec B}{A \prec \forall X.B}\ \textsc{Comp-AllR} \qquad\qquad \frac{A[X \mapsto \star] \prec B}{\forall X.A \prec B}\ \textsc{Comp-AllL}$$

Notice that, in rule Comp-AllL, the universal quantifier is *always* instantiated to $\star$. However, this way, $\lambda\mathsf{B}$ allows $\forall a.a \to a \prec \mathsf{Int} \to \mathsf{Bool}$, which as we discussed before might not be what we expect. Indeed $\lambda\mathsf{B}$ relies on sophisticated runtime checks to rule out such instances of the compatibility relation a posteriori.

$$\begin{tikzcd}
\bot \arrow[r, dash, "\sim"] & (\star \to \mathsf{Int}) \to \mathsf{Int} \\
(\forall a. a \to \mathsf{Int}) \to \mathsf{Int} \arrow[u, "<:"] \arrow[r, dash, "\sim"'] & (\forall a. \star \to \mathsf{Int}) \to \mathsf{Int} \arrow[u, "<:"]
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
\mathsf{Int} \to \mathsf{Int} \arrow[r, dash, "\sim"] & \mathsf{Int} \to \star \\
\forall a. a \arrow[u, "<:"] \arrow[r, dash, "\sim"'] & \bot \arrow[u, "<:"]
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
\bot \arrow[r, dash, "\sim"] & (((\star \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to \star) \\
(((\forall a. a \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\forall a. a) \arrow[u, "<:"] \arrow[r, dash, "\sim"'] & \bot \arrow[u, "<:"]
\end{tikzcd}$$

(c)

Fig. 4. Examples that break the original definition of consistent subtyping.

Igarashi et al. [14] introduced the so-called *quasi-polymorphic* types for types that may be used where a $\forall$-type is expected, which is important for their purpose of conservativity over System F. Their type consistency relation, involving polymorphism, is defined as follows[3]:

$$\frac{A \sim B}{\forall a.A \sim \forall a.B} \qquad \frac{A \sim B \qquad B \neq \forall a.B' \qquad \star \in \mathsf{Types}(B)}{\forall a.A \sim B}$$

Compared with our consistency definition in Fig. 3, their first rule is the same as ours. The second rule says that a non $\forall$-type can be consistent with a $\forall$-type only if it contains $\star$. In this way, their type system is able to reject $\forall a.a \to a \sim \mathsf{Int} \to \mathsf{Bool}$. However, in order to keep conservativity, they also reject $\forall a.a \to a \sim \mathsf{Int} \to \mathsf{Int}$, which is perfectly sensible in their setting (i.e., explicit polymorphism). However with implicit polymorphism, we would expect $\forall a.a \to a$ to be related with $\mathsf{Int} \to \mathsf{Int}$, since a can be instantiated to Int.

Nonetheless, when it comes to interactions between dynamically typed and polymorphically typed terms, both relations allow $\forall a.a \to \mathsf{Int}$ to be related with $\star \to \mathsf{Int}$ for example, which in our view, is some sort of (implicit) polymorphic subtyping combined with type consistency, and that should be derivable by the more primitive notions in the type system (instead of inventing new relations). One of our design principles is that subtyping and consistency is *orthogonal*, and can be naturally superimposed, echoing the same opinion of Siek and Taha [22].

3.2 Towards Consistent Subtyping

With the definitions of consistency and subtyping, the question now is how to compose these two relations so that two types can be compared in a way that takes these two relations into account.

[3] This is a simplified version.

Unfortunately, the original definition of Siek and Taha [22] (Definition 1) does not work well with our definitions of consistency and subtyping for polymorphic types. Consider two types: $(\forall a.a \to \mathsf{Int}) \to \mathsf{Int}$, and $(\star \to \mathsf{Int}) \to \mathsf{Int}$. The first type can only reach the second type in one way (first by applying consistency, then subtyping), but not the other way, as shown in Fig. 4a. We use $\bot$ to mean that we cannot find such a type. Similarly, there are situations where the first type can only reach the second type by the other way (first applying subtyping, and then consistency), as shown in Fig. 4b.

What is worse, if those two examples are composed in a way that those types all appear co-variantly, then the resulting types cannot reach each other in either way. For example, Fig. 4c shows such two types by putting a Bool type in the middle, and neither definition of consistent subtyping works.

Observations on Consistent Subtyping Based on Information Propagation. In order to develop the correct definition of consistent subtyping for polymorphic types, we need to understand how consistent subtyping works. We first review two important properties of subtyping: (1) subtyping induces the subsumption rule: if $A <: B$, then an expression of type A can be used where B is expected; (2) subtyping is transitive: if $A <: B$, and $B <: C$, then $A <: C$. Though consistent subtyping takes the unknown type into consideration, the subsumption rule should also apply: if $A \lesssim B$, then an expression of type A can also be used where B is expected, given that there might be some information lost by consistency. A crucial difference from subtyping is that consistent subtyping is *not* transitive because information can only be lost once (otherwise, any two types are a consistent subtype of each other). Now consider a situation where we have both $A <: B$, and $B \lesssim C$, this means that A can be used where B is expected, and B can be used where C is expected, with possibly some loss of information. In other words, we should expect that A can be used where C is expected, since there is at most one-time loss of information.

Observation 1. *If $A <: B$, and $B \lesssim C$, then $A \lesssim C$.*

This is reflected in Fig. 5a. A symmetrical observation is given in Fig. 5b:

Observation 2. *If $C \lesssim B$, and $B <: A$, then $C \lesssim A$.*

From the above observations, we see what the problem is with the original definition. In Fig. 5a, if B can reach C by T_1, then by subtyping transitivity, A can reach C by T_1. However, if B can only reach C by T_2, then A cannot reach C through the original definition. A similar problem is shown in Fig. 5b.

However, it turns out that those two problems can be fixed using the same strategy: instead of taking one-step subtyping and one-step consistency, our definition of consistent subtyping allows types to take *one-step subtyping, one-step consistency, and one more step subtyping*. Specifically, $A <: B \sim T_2 <: C$ (in Fig. 5a) and $C <: T_1 \sim B <: A$ (in Fig. 5b) have the same relation chain: subtyping, consistency, and subtyping.

$$\begin{tikzcd}
T_1 \arrow[r, dash, "\sim"] & C \\
B \arrow[u, "<:"] \arrow[ur, dashed, "\lesssim"] \arrow[r, dash, "\sim"'] & T_2 \arrow[u, "<:"'] \\
A \arrow[u, "<:"] \arrow[uur, dashed, bend right=60, "\lesssim"'] &
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
 & A \\
T_1 \arrow[r, dash, "\sim"] & B \arrow[u, "<:"'] \\
C \arrow[u, "<:"] \arrow[ur, dashed, "\lesssim"] \arrow[r, dash, "\sim"'] \arrow[uur, dashed, bend left=60, "\lesssim"] & T_2 \arrow[u, "<:"']
\end{tikzcd}$$

(b)

Fig. 5. Observations of consistent subtyping

$$\begin{tikzcd}
A_2 \arrow[r, dash, "\sim"] & A_3 \arrow[d, "<:"'] \\
A_1 \arrow[u, "<:"] \arrow[r, dashed, bend right, "\lesssim"] & A_4
\end{tikzcd}$$

$$\begin{aligned}
A_1 &= (((\forall a.a \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\forall a.a) \\
A_2 &= ((\forall a.a \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to \mathsf{Int}) \\
A_3 &= ((\forall a.\star \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to \star) \\
A_4 &= (((\star \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to \star)
\end{aligned}$$

Fig. 6. Example that is fixed by the new definition of consistent subtyping.

Definition of Consistent Subtyping. From the above discussion, we are ready to modify Definition 1, and adapt it to our notation:

Definition 2 (Consistent Subtyping)

$$\frac{\Psi \vdash A <: C \qquad C \sim D \qquad \Psi \vdash D <: B}{\Psi \vdash A \lesssim B}$$

With Definition 2, Fig. 6 illustrates the correct relation chain for the broken example shown in Fig. 4c. At first sight, Definition 2 seems worse than the original: we need to guess *two* types! It turns out that Definition 2 is a generalization of Definition 1, and they are equivalent in the system of Siek and Taha [22]. However, more generally, Definition 2 is compatible with polymorphic types.

Proposition 1 (Generalization of Consistent Subtyping)

- *Definition 2 subsumes Definition 1.*
- *Definition 1 is equivalent to Definition 2 in the system of Siek and Taha [22].*

3.3 Abstracting Gradual Typing

Garcia et al. [13] presented a new foundation for gradual typing that they call the *Abstracting Gradual Typing* (AGT) approach. In the AGT approach, gradual types are interpreted as sets of static types, where static types refer to types containing no unknown types. In this interpretation, predicates and functions on static types can then be lifted to apply to gradual types. Central to their approach is the so-called *concretization* function. For simple types, a concretization γ from gradual types to a set of static types[4] is defined as follows:

Definition 3 (Concretization)

$$\gamma(\mathsf{Int}) = \{\mathsf{Int}\} \qquad \gamma(A \to B) = \gamma(A) \to \gamma(B) \qquad \gamma(\star) = \{\textit{All static types}\}$$

Based on the concretization function, subtyping between static types can be lifted to gradual types, resulting in the consistent subtyping relation:

Definition 4 (Consistent Subtyping in AGT). *$A \mathrel{\widetilde{<:}} B$ if and only if $A_1 <: B_1$ for some $A_1 \in \gamma(A)$, $B_1 \in \gamma(B)$.*

Later they proved that this definition of consistent subtyping coincides with that of Siek and Taha [22] (Definition 1). By Proposition 1, we can directly conclude that our definition coincides with AGT:

Proposition 2 (Equivalence to AGT on Simple Types). *$A \lesssim B$ iff $A \mathrel{\widetilde{<:}} B$.*

However, AGT does not show how to deal with polymorphism (e.g. the interpretation of type variables) yet. Still, as noted by Garcia et al. [13], it is a promising line of future work for AGT, and the question remains whether our definition would coincide with it.

Another note related to AGT is that the definition is later adopted by Castagna and Lanvin [7], where the static types A_1, B_1 in Definition 4 can be algorithmically computed by also accounting for top and bottom types.

3.4 Directed Consistency

Directed consistency [15] is defined in terms of precision and static subtyping:

$$\frac{A' \sqsubseteq A \qquad A <: B \qquad B' \sqsubseteq B}{A' \lesssim B'}$$

The judgment $A \sqsubseteq B$ is read "A is less precise than B". In their setting, precision is defined for type constructors and subtyping for static types. If we interpret this definition from AGT's point of view, finding a more precise static type[5] has the same effect as concretization. Namely, $A' \sqsubseteq A$ implies $A \in \gamma(A')$ and $B' \sqsubseteq B$ implies $B \in \gamma(B')$. Therefore we consider this definition as AGT-style. From this perspective, this definition naturally coincides with Definition 2.

The value of their definition is that consistent subtyping is derived compositionally from *static subtyping* and *precision*. These are two more atomic relations. At first sight, their definition looks very similar to Definition 2 (replacing $\sqsubseteq$ by $<:$ and $<:$ by $\sim$). Then a question arises as to *which one is more fundamental*. To answer this, we need to discuss the relation between consistency and precision.

[4] For simplification, we directly regard type constructor $\to$ as a set-level operator.
[5] The definition of precision of types is given in appendix.

Relating Consistency and Precision. Precision is a partial order (anti-symmetric and transitive), while consistency is symmetric but not transitive. Nonetheless, precision and consistency are related by the following proposition:

Proposition 3 (Consistency and Precision)

– *If $A \sim B$, then there exists (static) C, such that $A \sqsubseteq C$, and $B \sqsubseteq C$.*
– *If for some (static) C, we have $A \sqsubseteq C$, and $B \sqsubseteq C$, then we have $A \sim B$.*

It may seem that precision is a more atomic relation, since consistency can be derived from precision. However, recall that consistency is in fact an equivalence relation lifted from static types to gradual types. Therefore defining consistency independently is straightforward, and it is theoretically viable to validate the definition of consistency directly. On the other hand, precision is usually connected with the gradual criteria [25], and finding a correct partial order that adheres to the criteria is not always an easy task. For example, Igarashi et al. [14] argued that term precision for System F_G is actually nontrivial, leaving the gradual guarantee of the semantics as a conjecture. Thus precision can be difficult to extend to more sophisticated type systems, e.g. dependent types.

Still, it is interesting that those two definitions illustrate the correspondence of different foundations (on simple types): one is defined directly on gradual types, and the other stems from AGT, which is based on static subtyping.

3.5 Consistent Subtyping Without Existentials

Definition 2 serves as a fine specification of how consistent subtyping should behave in general. But it is inherently non-deterministic because of the two intermediate types C and D. As with Definition 1, we need a combined relation to directly compare two types. A natural attempt is to try to extend the restriction operator for polymorphic types. Unfortunately, as we show below, this does not work. However it is possible to devise an equivalent inductive definition instead.

Attempt to Extend the Restriction Operator. Suppose that we try to extend the restriction operator to account for polymorphic types. The original restriction operator is structural, meaning that it works for types of similar structures. But for polymorphic types, two input types could have different structures due to universal quantifiers, e.g., $\forall a.a \to \mathsf{Int}$ and $(\mathsf{Int} \to \star) \to \mathsf{Int}$. If we try to mask the first type using the second, it seems hard to maintain the information that a should be instantiated to a function while ensuring that the return type is masked. There seems to be no satisfactory way to extend the restriction operator in order to support this kind of non-structural masking.

Interpretation of the Restriction Operator and Consistent Subtyping. If the restriction operator cannot be extended naturally, it is useful to take a step back and revisit what the restriction operator actually does. For consistent subtyping, two input types could have unknown types in different positions, but we only care about the known parts. What the restriction operator does is (1) erase

$$\boxed{\Psi \vdash A \lesssim B}$$

$$\frac{\Psi, a \vdash A \lesssim B}{\Psi \vdash A \lesssim \forall a.B}\ \textsc{CS-ForallR} \qquad \frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] \lesssim B}{\Psi \vdash \forall a.A \lesssim B}\ \textsc{CS-ForallL}$$

$$\frac{\Psi \vdash B_1 \lesssim A_1 \qquad \Psi \vdash A_2 \lesssim B_2}{\Psi \vdash A_1 \to A_2 \lesssim B_1 \to B_2}\ \textsc{CS-Fun} \qquad \frac{a \in \Psi}{\Psi \vdash a \lesssim a}\ \textsc{CS-TVar} \qquad \frac{}{\Psi \vdash \mathsf{Int} \lesssim \mathsf{Int}}\ \textsc{CS-Int}$$

$$\frac{}{\Psi \vdash \star \lesssim A}\ \textsc{CS-UnknownL} \qquad \frac{}{\Psi \vdash A \lesssim \star}\ \textsc{CS-UnknownR}$$

Fig. 7. Consistent Subtyping for implicit polymorphism.

the type information in one type if the corresponding position in the other type is the unknown type; and (2) compare the resulting types using the normal subtyping relation. The example below shows the masking-off procedure for the types $\mathsf{Int} \to \star \to \mathsf{Bool}$ and $\mathsf{Int} \to \mathsf{Int} \to \star$. Since the known parts have the relation that $\mathsf{Int} \to \star \to \star <: \mathsf{Int} \to \star \to \star$, we conclude that $\mathsf{Int} \to \star \to \mathsf{Bool} \lesssim \mathsf{Int} \to \mathsf{Int} \to \star$.

$$\left.\begin{array}{l} \mathsf{Int} \to \boxed{\star} \to \boxed{\mathsf{Bool}}\ |_{\mathsf{Int} \to \mathsf{Int} \to \star} = \mathsf{Int} \to \star \to \star \\ \mathsf{Int} \to \boxed{\mathsf{Int}} \to \boxed{\star}\ |_{\mathsf{Int} \to \star \to \mathsf{Bool}} = \mathsf{Int} \to \star \to \star \end{array}\right\} <:$$

Here differences of the types in boxes are erased because of the restriction operator. Now if we compare the types in boxes directly instead of through the lens of the restriction operator, we can observe that the *consistent subtyping relation always holds between the unknown type and an arbitrary type.* We can interpret this observation directly from Definition 2: the unknown type is neutral to subtyping ($\star <: \star$), the unknown type is consistent with any type ($\star \sim A$), and subtyping is reflexive ($A <: A$). Therefore, *the unknown type is a consistent subtype of any type ($\star \lesssim A$), and vice versa ($A \lesssim \star$).* Note that this interpretation provides a general recipe on how to lift a (static) subtyping relation to a (gradual) consistent subtyping relation, as discussed below.

Defining Consistent Subtyping Directly. From the above discussion, we can define the consistent subtyping relation directly, *without* resorting to subtyping or consistency at all. The key idea is that we replace $<:$ with $\lesssim$ in Fig. 3, get rid of rule S-Unknown and add two extra rules concerning $\star$, resulting in the rules of consistent subtyping in Fig. 7. Of particular interest are the rules CS-UnknownL and CS-UnknownR, both of which correspond to what we just said: the unknown type is a consistent subtype of any type, and vice versa. From now on, we use the symbol $\lesssim$ to refer to the consistent subtyping relation in Fig. 7. What is more, we can prove that those two are equivalent[6]:

$\mathcal{T}$heorem 1. $\Psi \vdash A \lesssim B \Leftrightarrow \Psi \vdash A <: C$, $C \sim D$, $\Psi \vdash D <: B$ *for some* C, D.

[6] Theorems with $\mathcal{T}$ are those proved in Coq. The same applies to $\mathcal{L}$emmas.

$$\boxed{\Psi \vdash e : A \rightsquigarrow s}$$

$$\frac{x : A \in \Psi}{\Psi \vdash x : A \rightsquigarrow x}\ \textsc{Var} \qquad \frac{}{\Psi \vdash n : \mathsf{Int} \rightsquigarrow n}\ \textsc{Nat} \qquad \frac{\Psi, a \vdash e : A \rightsquigarrow s}{\Psi \vdash e : \forall a.A \rightsquigarrow \Lambda a.s}\ \textsc{Gen}$$

$$\frac{\Psi, x : A \vdash e : B \rightsquigarrow s}{\Psi \vdash \lambda x : A.\ e : A \to B \rightsquigarrow \lambda x : A.\ s}\ \textsc{LamAnn} \qquad \frac{\Psi, x : \tau \vdash e : B \rightsquigarrow s}{\Psi \vdash \lambda x.\ e : \tau \to B \rightsquigarrow \lambda x : \tau.\ s}\ \textsc{Lam}$$

$$\frac{\Psi \vdash e_1 : A \rightsquigarrow s_1 \qquad \Psi \vdash A \rhd A_1 \to A_2 \qquad \Psi \vdash e_2 : A_3 \rightsquigarrow s_2 \qquad \Psi \vdash A_3 \lesssim A_1}{\Psi \vdash e_1\ e_2 : A_2 \rightsquigarrow (\langle A \hookrightarrow A_1 \to A_2 \rangle\ s_1)\ (\langle A_3 \hookrightarrow A_1 \rangle\ s_2)}\ \textsc{App}$$

$$\boxed{\Psi \vdash A \rhd A_1 \to A_2}$$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] \rhd A_1 \to A_2}{\Psi \vdash \forall a.A \rhd A_1 \to A_2}\ \textsc{M-Forall}$$

$$\frac{}{\Psi \vdash (A_1 \to A_2) \rhd (A_1 \to A_2)}\ \textsc{M-Arr} \qquad \frac{}{\Psi \vdash \star \rhd \star \to \star}\ \textsc{M-Unknown}$$

Fig. 8. Declarative typing

4 Gradually Typed Implicit Polymorphism

In Sect. 3 we introduced the consistent subtyping relation that accommodates polymorphic types. In this section we continue with the development by giving a declarative type system for predicative implicit polymorphism that employs the consistent subtyping relation. The declarative system itself is already quite interesting as it is equipped with both higher-rank polymorphism and the unknown type. The syntax of expressions in the declarative system is given below:

$$\text{Expressions} \quad e ::= x \mid n \mid \lambda x : A.\ e \mid \lambda x.\ e \mid e\ e$$

4.1 Typing in Detail

Figure 8 gives the typing rules for our declarative system (the reader is advised to ignore the gray-shaded parts for now). Rule Var extracts the type of the variable from the typing context. Rule Nat always infers integer types. Rule LamAnn puts x with type annotation A into the context, and continues type checking the body e. Rule Lam assigns a monotype τ to x, and continues type checking the body e. Gradual types and polymorphic types are introduced via annotations explicitly. Rule Gen puts a fresh type variable a into the type context and generalizes the typing result A to $\forall a.A$. Rule App first infers the type of e_1, then the matching judgment $\Psi \vdash A \rhd A_1 \to A_2$ extracts the domain type A_1 and the codomain type A_2 from type A. The type A_3 of the argument e_2 is then compared with A_1 using the consistent subtyping judgment.

Matching. The matching judgment of Siek et al. [25] can be extended to polymorphic types naturally, resulting in $\Psi \vdash A \triangleright A_1 \rightarrow A_2$. In M-Forall, a monotype τ is guessed to instantiate the universal quantifier a. This rule is inspired by the *application judgment* $\Phi \vdash A \bullet e \Rightarrow C$ [11], which says that if we apply a term of type A to an argument e, we get something of type C. If A is a polymorphic type, the judgment works by guessing instantiations until it reaches an arrow type. Matching further simplifies the application judgment, since it is independent of typing. Rule M-Arr and M-Unknown are the same as Siek et al. [25]. M-Arr returns the domain type A_1 and range type A_2 as expected. If the input is $\star$, then M-Unknown returns $\star$ as both the type for the domain and the range.

Note that matching saves us from having a subsumption rule (Sub in Fig. 2). the subsumption rule is incompatible with consistent subtyping, since the latter is not transitive. A discussion of a subsumption rule based on normal subtyping can be found in the appendix.

4.2 Type-Directed Translation

We give the dynamic semantics of our language by translating it to $\lambda\mathsf{B}$. Below we show a subset of the terms in $\lambda\mathsf{B}$ that are used in the translation:

$$\text{Terms} \quad s ::= x \mid n \mid \lambda x : A.\ s \mid \Lambda a.s \mid s_1\ s_2 \mid \langle A \hookrightarrow B \rangle\ s$$

A cast $\langle A \hookrightarrow B \rangle\ s$ converts the value of term s from type A to type B. A cast from A to B is permitted only if the types are *compatible*, written $A \prec B$, as briefly mentioned in Sect. 3.1. The syntax of types in $\lambda\mathsf{B}$ is the same as ours.

The translation is given in the gray-shaded parts in Fig. 8. The only interesting case here is to insert explicit casts in the application rule. Note that there is no need to translate matching or consistent subtyping, instead we insert the source and target types of a cast directly in the translated expressions, thanks to the following two lemmas:

Lemma 1 ($\triangleright$ to $\prec$). *If $\Psi \vdash A \triangleright A_1 \rightarrow A_2$, then $A \prec A_1 \rightarrow A_2$.*

Lemma 2 ($\lesssim$ to $\prec$). *If $\Psi \vdash A \lesssim B$, then $A \prec B$.*

In order to show the correctness of the translation, we prove that our translation always produces well-typed expressions in $\lambda\mathsf{B}$. By Lemmas 1 and 2, we have the following theorem:

Theorem 2 (Type Safety). *If $\Psi \vdash e : A \rightsquigarrow s$, then $\Psi \vdash^B s : A$.*

Parametricity. An important semantic property of polymorphic types is *relational parametricity* [19]. The parametricity property says that all instances of a polymorphic function should behave *uniformly.* A classic example is a function with the type $\forall a.a \rightarrow a$. The parametricity property guarantees that a value of this type must be either the identity function (i.e., $\lambda x.x$) or the undefined function (one which never returns a value). However, with the addition of the unknown type $\star$, careful measures are to be taken to ensure parametricity. This is exactly the circumstance that $\lambda\mathsf{B}$ was designed to address. Ahmed et al. [2] proved that $\lambda\mathsf{B}$ satisfies relational parametricity. Based on their result, and by Theorem 2, parametricity is preserved in our system.

Ambiguity from Casts. The translation does not always produce a unique target expression. This is because when we guess a monotype τ in rule M-Forall and CS-ForallL, we could have different choices, which inevitably leads to different types. Unlike (non-gradual) polymorphic type systems [11,18], the choice of monotypes could affect runtime behaviour of the translated programs, since they could appear inside the explicit casts. For example, the following shows two possible translations for the same source expression $\lambda x : \star.\ f\ x$, where the type of f is instantiated to $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Bool}$, respectively:

$$
\begin{aligned}
&f : \forall a.a \to a \vdash (\lambda x : \star.\ f\ x) : \star \to \mathsf{Int} \\
&\qquad\qquad \rightsquigarrow (\lambda x : \star.\ (\langle \forall a.a \to a \hookrightarrow \mathsf{Int} \to \mathsf{Int} \rangle\ f)\ (\ \langle \star \hookrightarrow \mathsf{Int} \rangle\ \ x)) \\
&f : \forall a.a \to a \vdash (\lambda x : \star.\ f\ x) : \star \to \mathsf{Bool} \\
&\qquad\qquad \rightsquigarrow (\lambda x : \star.\ (\langle \forall a.a \to a \hookrightarrow \mathsf{Bool} \to \mathsf{Bool} \rangle\ f)\ (\ \langle \star \hookrightarrow \mathsf{Bool} \rangle\ \ x))
\end{aligned}
$$

If we apply $\lambda x : \star.\ f\ x$ to 3, which is fine since the function can take any input, the first translation runs smoothly in $\lambda\mathsf{B}$, while the second one will raise a cast error (Int cannot be cast to Bool). Similarly, if we apply it to true, then the second succeeds while the first fails. The culprit lies in the highlighted parts where any instantiation of a would be put inside the explicit cast. More generally, any choice introduces an explicit cast to that type in the translation, which causes a runtime cast error if the function is applied to a value whose type does not match the guessed type. Note that this does not compromise the type safety of the translated expressions, since cast errors are part of the type safety guarantees.

Coherence. The ambiguity of translation seems to imply that the declarative system is *incoherent*. A semantics is coherent if distinct typing derivations of the same typing judgment possess the same meaning [20]. We argue that the declarative system is "coherent up to cast errors" in the sense that a well-typed program produces a unique value, or results in a cast error. In the above example, whatever the translation might be, applying $\lambda x : \star.\ f\ x$ to 3 either results in a cast error, or produces 3, nothing else.

This discrepancy is due to the guessing nature of the *declarative* system. As far as the declarative system is concerned, both $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Bool}$ are equally acceptable. But this is not the case at runtime. The acute reader may have found that the *only* appropriate choice is to instantiate f to $\star \to \star$. However, as specified by rule M-Forall in Fig. 8, we can only instantiate type variables to monotypes, but $\star$ is *not* a monotype! We will get back to this issue in Sect. 6.2 after we present the corresponding algorithmic system in Sect. 5.

4.3 Correctness Criteria

Siek et al. [25] present a set of properties that a well-designed gradual typing calculus must have, which they call the refined criteria. Among all the criteria, those related to the static aspects of gradual typing are well summarized by Cimini and Siek [8]. Here we review those criteria and adapt them to our notation. We have proved in Coq that our type system satisfies all these criteria.

Lemma 3 (Correctness Criteria)

- ***Conservative extension:*** *for all static Ψ, e, and A,*
  - *if $\Psi \vdash^{OL} e : A$, then there exists B, such that $\Psi \vdash e : B$, and $\Psi \vdash B <: A$.*
  - *if $\Psi \vdash e : A$, then $\Psi \vdash^{OL} e : A$*
- ***Monotonicity w.r.t. precision:*** *for all Ψ, e, e', A, if $\Psi \vdash e : A$, and $e' \sqsubseteq e$, then $\Psi \vdash e' : B$, and $B \sqsubseteq A$ for some B.*
- ***Type Preservation of cast insertion:*** *for all Ψ, e, A, if $\Psi \vdash e : A$, then $\Psi \vdash e : A \rightsquigarrow s$, and $\Psi \vdash^{B} s : A$ for some s.*
- ***Monotonicity of cast insertion:*** *for all $\Psi, e_1, e_2, e_1', e_2', A$, if $\Psi \vdash e_1 : A \rightsquigarrow e_1'$, and $\Psi \vdash e_2 : A \rightsquigarrow e_2'$, and $e_1 \sqsubseteq e_2$, then $\Psi \mid \Psi \vdash e_1' \sqsubseteq^{B} e_2'$.*

The first criterion states that the gradual type system should be a conservative extension of the original system. In other words, a *static* program that is typeable in the Odersky-Läufer type system if and only if it is typeable in the gradual type system. A static program is one that does not contain any type $\star$[7]. However since our gradual type system does not have the subsumption rule, it produces more general types.

The second criterion states that if a typeable expression loses some type information, it remains typeable. This criterion depends on the definition of the precision relation, written $A \sqsubseteq B$, which is given in the appendix. The relation intuitively captures a notion of types containing more or less unknown types ($\star$). The precision relation over types lifts to programs, i.e., $e_1 \sqsubseteq e_2$ means that e_1 and e_2 are the same program except that e_2 has more unknown types.

The first two criteria are fundamental to gradual typing. They explain for example why these two programs ($\lambda x : \mathsf{Int}.\ x + 1$) and ($\lambda x : \star.\ x + 1$) are typeable, as the former is typeable in the Odersky-Läufer type system and the latter is a less-precise version of it.

The last two criteria relate the compilation to the cast calculus. The third criterion is essentially the same as Theorem 2, given that a target expression should always exist, which can be easily seen from Fig. 8. The last criterion ensures that the translation must be monotonic over the precision relation $\sqsubseteq$.

As for the dynamic guarantee, things become a bit murky for two reasons: (1) as we discussed before, our declarative system is incoherent in that the runtime behaviour of the same source program can vary depending on the particular translation; (2) it is still unknown whether dynamic guarantee holds in $\lambda\mathsf{B}$. We will have more discussion on the dynamic guarantee in Sect. 6.3.

5 Algorithmic Type System

In this section we give a bidirectional account of the algorithmic type system that implements the declarative specification. The algorithm is largely inspired by the

[7] Note that the term *static* has appeared several times with different meanings.

Expressions	$e ::= x \mid n \mid \lambda x : A.\ e \mid \lambda x.\ e \mid e\ e \mid e : A$
Types	$A, B ::= \mathsf{Int} \mid a \mid \widehat{a} \mid A \rightarrow B \mid \forall a.A \mid \star$
Monotypes	$\tau, \sigma ::= \mathsf{Int} \mid a \mid \widehat{a} \mid \tau \rightarrow \sigma$
Contexts	$\Gamma, \Delta, \Theta ::= \varnothing \mid \Gamma, x : A \mid \Gamma, a \mid \Gamma, \widehat{a} \mid \Gamma, \widehat{a} = \tau$
Complete Contexts	$\Omega ::= \varnothing \mid \Omega, x : A \mid \Omega, a \mid \Omega, \widehat{a} = \tau$

Fig. 9. Syntax of the algorithmic system

$$\boxed{\Gamma \vdash A \lesssim B \dashv \Delta}$$

$$\frac{}{\Gamma[a] \vdash a \lesssim a \dashv \Gamma[a]}\ \text{ACS-TVar} \qquad \frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \lesssim \widehat{a} \dashv \Gamma[\widehat{a}]}\ \text{ACS-ExVar}$$

$$\frac{}{\Gamma \vdash \mathsf{Int} \lesssim \mathsf{Int} \dashv \Gamma}\ \text{ACS-Int} \qquad \frac{}{\Gamma \vdash \star \lesssim A \dashv \Gamma}\ \text{ACS-UnknownL} \qquad \frac{}{\Gamma \vdash A \lesssim \star \dashv \Gamma}\ \text{ACS-UnknownR}$$

$$\frac{\Gamma \vdash B_1 \lesssim A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \lesssim [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \rightarrow A_2 \lesssim B_1 \rightarrow B_2 \dashv \Delta}\ \text{ACS-Fun}$$

$$\frac{\Gamma, a \vdash A \lesssim B \dashv \Delta, a, \Theta}{\Gamma \vdash A \lesssim \forall a.B \dashv \Delta}\ \text{ACS-ForallR} \qquad \frac{\Gamma, \widehat{a} \vdash A[a \mapsto \widehat{a}] \lesssim B \dashv \Delta}{\Gamma \vdash \forall a.A \lesssim B \dashv \Delta}\ \text{ACS-ForallL}$$

$$\frac{\widehat{a} \notin fv(A) \qquad \Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox A \dashv \Delta}{\Gamma[\widehat{a}] \vdash \widehat{a}. \lesssim A \dashv \Delta}\ \text{ACS-InstL} \qquad \frac{\widehat{a} \notin fv(A) \qquad \Gamma[\widehat{a}] \vdash A \lessapprox \widehat{a} \dashv \Delta}{\Gamma[\widehat{a}] \vdash A \lesssim \widehat{a} \dashv \Delta}\ \text{ACS-InstR}$$

Fig. 10. Algorithmic consistent subtyping

algorithmic bidirectional system of Dunfield and Krishnaswami [11] (henceforth DK system). However our algorithmic system differs from theirs in three aspects: (1) the addition of the unknown type $\star$; (2) the use of the matching judgment; and (3) the approach of *gradual inference only producing static types* [12]. We then prove that our algorithm is both sound and complete with respect to the declarative type system. Full proofs can be found in the appendix.

Algorithmic Contexts. The algorithmic context Γ is an *ordered* list containing declarations of type variables a and term variables $x : A$. Unlike declarative contexts, algorithmic contexts also contain declarations of existential type variables $\widehat{a}$, which can be either unsolved (written $\widehat{a}$) or solved to some monotype (written $\widehat{a} = \tau$). Complete contexts Ω are those that contain no unsolved existential type variables. Figure 9 shows the syntax of the algorithmic system. Apart from expressions in the declarative system, we have annotated expressions $e : A$.

5.1 Algorithmic Consistent Subtyping and Instantiation

Figure 10 shows the algorithmic consistent subtyping rules. The first five rules do not manipulate contexts. Rule ACS-Fun is a natural extension of its declarative counterpart. The output context of the first premise is used by the second

$$\boxed{\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta}$$

$$\frac{\Gamma \vdash \tau}{\Gamma, \widehat{a}, \Gamma' \vdash \widehat{a} \lessapprox \tau \dashv \Gamma, \widehat{a} = \tau, \Gamma'}\ \textsc{InstLSolve} \qquad \frac{}{\Gamma[\widehat{a}][\widehat{b}] \vdash \widehat{a} \lessapprox \widehat{b} \dashv \Gamma[\widehat{a}][\widehat{b} = \widehat{a}]}\ \textsc{InstLReach}$$

$$\frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox \star \dashv \Gamma[\widehat{a}]}\ \textsc{InstLSolveU} \qquad \frac{\Gamma[\widehat{a}], b \vdash \widehat{a} \lessapprox B \dashv \Delta, b, \Delta'}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox \forall b.B \dashv \Delta}\ \textsc{InstLAllR}$$

$$\frac{\Gamma[\widehat{a}_2, \widehat{a}_1, \widehat{a} = \widehat{a}_1 \rightarrow \widehat{a}_2] \vdash A_1 \lessapprox \widehat{a}_1 \dashv \Theta \qquad \Theta \vdash \widehat{a}_2 \lessapprox [\Theta]A_2 \dashv \Delta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox A_1 \rightarrow A_2 \dashv \Delta}\ \textsc{InstLArr}$$

Fig. 11. Algorithmic instantiation

premise, and the output context of the second premise is the output context of the conclusion. Note that we do not simply check $A_2 \lessapprox B_2$, but apply Θ to both types (e.g., $[\Theta]A_2$). This is to maintain an important invariant that types are fully applied under input context Γ (they contain no existential variables already solved in Γ). The same invariant applies to every algorithmic judgment. Rule ACS-ForallR looks similar to its declarative counterpart, except that we need to drop the trailing context a, Θ from the concluding output context since they become out of scope. Rule ACS-ForallL generates a fresh existential variable $\widehat{a}$, and replaces a with $\widehat{a}$ in the body A. The new existential variable $\widehat{a}$ is then added to the premise's input context. As a side note, when both types are quantifiers, then either ACS-ForallR or ACS-ForallR could be tried. In practice, one can apply ACS-ForallR eagerly. The last two rules together check consistent subtyping with an unsolved existential variable on one side and an arbitrary type on the other side by the help of the instantiation judgment.

The judgment $\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta$ defined in Fig. 11 instantiates unsolved existential variables. Judgment $\widehat{a} \lessapprox A$ reads "instantiate $\widehat{a}$ to a consistent subtype of A". For space reasons, we omit its symmetric judgement $\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta$. Rule InstLSolve and rule InstLReach set $\widehat{a}$ to τ and $\widehat{b}$ in the output context, respectively. Rule InstLSolveU is similar to ACS-UnknownR in that we put no constraint on $\widehat{a}$ when it meets the unknown type $\star$. This design decision reflects the point that type inference only produces static types [12]. We will get back to this point in Sect. 6.2. Rule InstLAllR is the instantiation version of rule ACS-ForallR. The last rule InstLArr applies when $\widehat{a}$ meets a function type. It follows that the solution must also be a function type. That is why, in the first premise, we generate two fresh existential variables $\widehat{a}_1$ and $\widehat{a}_2$, and insert them just before $\widehat{a}$ in the input context, so that the solution of $\widehat{a}$ can mention them. Note that $A_1 \lessapprox \widehat{a}_1$ switches to the other instantiation judgment.

5.2 Algorithmic Typing

We now turn to the algorithmic typing rules in Fig. 12. The algorithmic system uses bidirectional type checking to accommodate polymorphism. Most of

$$\boxed{\Gamma \vdash e \Rightarrow A \dashv \Delta}$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \dashv \Gamma}\,\textsc{AVar} \qquad \frac{}{\Gamma \vdash n \Rightarrow \mathsf{Int} \dashv \Gamma}\,\textsc{ANat}$$

$$\frac{\Gamma, \widehat{a}, \widehat{b}, x : \widehat{a} \vdash e \Leftarrow \widehat{b} \dashv \Delta, x : \widehat{a}, \Theta}{\Gamma \vdash \lambda x.\ e \Rightarrow \widehat{a} \to \widehat{b} \dashv \Delta}\,\textsc{ALamU} \qquad \frac{\Gamma, x : A \vdash e \Rightarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x : A.\ e \Rightarrow A \to B \dashv \Delta}\,\textsc{ALamAnnA}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash e \Leftarrow A \dashv \Delta}{\Gamma \vdash e : A \Rightarrow A \dashv \Delta}\,\textsc{AAnno}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta_1 \qquad \Theta_1 \vdash [\Theta_1]A \rhd A_1 \to A_2 \dashv \Theta_2 \qquad \Theta_2 \vdash e_2 \Leftarrow [\Theta_2]A_1 \dashv \Delta}{\Gamma \vdash e_1\ e_2 \Rightarrow A_2 \dashv \Delta}\,\textsc{AApp}$$

$$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$$

$$\frac{\Gamma, x : A \vdash e \Leftarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x.\ e \Leftarrow A \to B \dashv \Delta}\,\textsc{ALam} \qquad \frac{\Gamma, a \vdash e \Leftarrow A \dashv \Delta, a, \Theta}{\Gamma \vdash e \Leftarrow \forall a.A \dashv \Delta}\,\textsc{AGen}$$

$$\frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A \lesssim [\Theta]B \dashv \Delta}{\Gamma \vdash e \Leftarrow B \dashv \Delta}\,\textsc{ASub}$$

$$\boxed{\Gamma \vdash A \rhd A_1 \to A_2 \dashv \Delta}$$

$$\frac{\Gamma, \widehat{a} \vdash A[a \mapsto \widehat{a}] \rhd A_1 \to A_2 \dashv \Delta}{\Gamma \vdash \forall a.A \rhd A_1 \to A_2 \dashv \Delta}\,\textsc{AM-Forall} \qquad \frac{}{\Gamma \vdash (A_1 \to A_2) \rhd (A_1 \to A_2) \dashv \Gamma}\,\textsc{AM-Arr}$$

$$\frac{}{\Gamma \vdash \star \rhd \star \to \star \dashv \Gamma}\,\textsc{AM-Unknown} \qquad \frac{}{\Gamma[\widehat{c}] \vdash \widehat{c} \rhd \widehat{a} \to \widehat{b} \dashv \Gamma[\widehat{a}, \widehat{b}, \widehat{c} = \widehat{a} \to \widehat{b}]}\,\textsc{AM-Var}$$

Fig. 12. Algorithmic typing

them are quite standard. Perhaps rule AApp (which differs significantly from that in the DK system) deserves attention. It relies on the algorithmic matching judgment $\Gamma \vdash A \rhd A_1 \to A_2 \dashv \Delta$. Rule AM-ForallL replaces a with a fresh existential variable $\widehat{a}$, thus eliminating guessing. Rule AM-Arr and AM-Unknown correspond directly to the declarative rules. Rule AM-Var, which has no corresponding declarative version, is similar to InstRArr/InstLArr: we create $\widehat{a}$ and $\widehat{b}$ and add $\widehat{c} = \widehat{a} \to \widehat{b}$ to the context.

5.3 Completeness and Soundness

We prove that the algorithmic rules are sound and complete with respect to the declarative specifications. We need an auxiliary judgment $\Gamma \longrightarrow \Delta$ that captures a notion of information increase from input contexts Γ to output contexts Δ [11].

Soundness. Roughly speaking, soundness of the algorithmic system says that given an expression e that type checks in the algorithmic system, there exists a corresponding expression e' that type checks in the declarative system. However there is one complication: e does not necessarily have more annotations than e'. For example, by ALam we have $\lambda x.\ x \Leftarrow (\forall a.a) \rightarrow (\forall a.a)$, but $\lambda x.\ x$ itself cannot have type $(\forall a.a) \rightarrow (\forall a.a)$ in the declarative system. To circumvent that, we add an annotation to the lambda abstraction, resulting in $\lambda x : (\forall a.a).\ x$, which is typeable in the declarative system with the same type. To relate $\lambda x.\ x$ and $\lambda x : (\forall a.a).\ x$, we erase all annotations on both expressions. The definition of erasure $\lfloor \cdot \rfloor$ is standard and thus omitted.

Theorem 1 (Soundness of Algorithmic Typing). *Given $\Delta \longrightarrow \Omega$,*

1. *If $\Gamma \vdash e \Rightarrow A \dashv \Delta$ then $\exists e'$ such that $[\Omega]\Delta \vdash e' : [\Omega]A$ and $\lfloor e \rfloor = \lfloor e' \rfloor$.*
2. *If $\Gamma \vdash e \Leftarrow A \dashv \Delta$ then $\exists e'$ such that $[\Omega]\Delta \vdash e' : [\Omega]A$ and $\lfloor e \rfloor = \lfloor e' \rfloor$.*

Completeness. Completeness of the algorithmic system is the reverse of soundness: given a declarative judgment of the form $[\Omega]\Gamma \vdash [\Omega] \ldots$, we want to get an algorithmic derivation of $\Gamma \vdash \cdots \dashv \Delta$. It turns out that completeness is a bit trickier to state in that the algorithmic rules generate existential variables on the fly, so Δ could contain unsolved existential variables that are not found in Γ, nor in Ω. Therefore the completeness proof must produce another complete context Ω' that extends both the output context Δ, and the given complete context Ω. As with soundness, we need erasure to relate both expressions.

Theorem 2 (Completeness of Algorithmic Typing). *Given $\Gamma \longrightarrow \Omega$ and $\Gamma \vdash A$, if $[\Omega]\Gamma \vdash e : A$ then there exist Δ, Ω', A' and e' such that $\Delta \longrightarrow \Omega'$ and $\Omega \longrightarrow \Omega'$ and $\Gamma \vdash e' \Rightarrow A' \dashv \Delta$ and $A = [\Omega']A'$ and $\lfloor e \rfloor = \lfloor e' \rfloor$.*

6 Discussion

6.1 Top Types

To demonstrate that our definition of consistent subtyping (Definition 2) is applicable to other features, we show how to extend our approach to Top types with all the desired properties preserved.

In order to preserve the orthogonality between subtyping and consistency, we require $\top$ to be a common supertype of all static types, as shown in rule S-Top. This rule might seem strange at first glance, since even if we remove the requirement *A static*, the rule seems reasonable. However, an important point is that because of the orthogonality between subtyping and consistency, subtyping itself should not contain a potential information loss! Therefore, subtyping instances such as $\star <: \top$ are not allowed. For consistency, we add the rule that $\top$ is consistent with $\top$, which is actually included in the original reflexive rule

$A \sim A$. For consistent subtyping, every type is a consistent subtype of $\top$, for example, $\mathsf{Int} \to \star \lesssim \top$.

$$\frac{A \textit{ static}}{\Psi \vdash A <: \top}\ \text{S-Top} \qquad \top \sim \top \qquad \frac{}{\Psi \vdash A \lesssim \top}\ \text{CS-Top}$$

It is easy to verify that Definition 2 is still equivalent to that in Fig. 7 extended with rule CS-Top. That is, Theorem 1 holds:

Proposition 4 (Extension with $\top$). *$\Psi \vdash A \lesssim B \Leftrightarrow \Psi \vdash A <: C$, $C \sim D$, $\Psi \vdash D <: B$, for some C, D.*

We extend the definition of concretization (Definition 3) with $\top$ by adding another equation $\gamma(\top) = \{\top\}$. Note that Castagna and Lanvin [7] also have this equation in their calculus. It is easy to verify that Proposition 2 still holds:

Proposition 5 (Equivalent to AGT on $\top$). *$A \lesssim B$ if only if $A \mathrel{\widetilde{<:}} B$.*

Siek and Taha's [22] *Definition of Consistent Subtyping Does Not Work for* $\top$. As the analysis in Sect. 3.2, $\mathsf{Int} \to \star \lesssim \top$ only holds when we first apply consistency, then subtyping. However we cannot find a type A such that $\mathsf{Int} \to \star <: A$ and $A \sim \top$. Also we have a similar problem in extending the restriction operator: *non-structural* masking between $\mathsf{Int} \to \star$ and $\top$ cannot be easily achieved.

6.2 Interpretation of the Dynamic Semantics

In Sect. 4.2 we have seen an example where a source expression could produce two different target expressions with different runtime behaviour. As we explained, this is due to the guessing nature of the declarative system, and from the typing point of view, no type is particularly better than others. However, in practice, this is not desirable. Let us revisit the same example, now from the algorithmic point of view (we omit the translation for space reasons):

$$f : \forall a.a \to a \vdash (\lambda x : \star.\ f\ x) \Rightarrow \star \to \widehat{a} \dashv f : \forall a.a \to a, \widehat{a}$$

Compared with declarative typing, which produces many types ($\star \to \mathsf{Int}$, $\star \to \mathsf{Bool}$, and so on), the algorithm computes the type $\star \to \widehat{a}$ with $\widehat{a}$ unsolved in the output context. What can we know from the output context? The only thing we know is that $\widehat{a}$ is not constrained at all! However, it is possible to make a more refined distinction between different kinds of existential variables. The first kind of existential variables are those that indeed have no constraints at all, as they do not affect the dynamic semantics. The second kind of existential variables (as in this example) are those where the only constraint is that *the variable was once compared with an unknown type* [12].

To emphasize the difference and have better support for dynamic semantics, we could have *gradual variables* in addition to existential variables, with the difference that only unsolved gradual variables are allowed to be unified with the unknown type. An irreversible transition from existential variables to gradual variables occurs when an existential variable is compared with $\star$. After the algorithm terminates, we can set all unsolved existential variables to be any (static) type (or more precisely, as Garcia and Cimini [12], with *static type parameters*), and all unsolved gradual variables to be $\star$ (or *gradual type parameters*). However, this approach requires a more sophisticated declarative/algorithmic type system than the ones presented in this paper, where we only produce static monotypes in type inference. We believe this is a typical trade-off in existing gradual type systems with inference [12,23]. Here we suppress the complexity of dynamic semantics in favour of the conciseness of static typing.

6.3 The Dynamic Guarantee

In Sect. 4.3 we mentioned that the dynamic guarantee is closely related to the coherence issue. To aid discussion, we first give the definition of dynamic guarantee as follows:

Definition 5 (Dynamic guarantee). *Suppose* $e' \sqsubseteq e$, $\emptyset \vdash e : A \rightsquigarrow s$ *and* $\emptyset \vdash e' : A' \rightsquigarrow s'$, *if* $s \Downarrow v$, *then* $s' \Downarrow v'$ *and* $v' \sqsubseteq v$.

The dynamic guarantee says that if a gradually typed program evaluates to a value, then removing type annotations always produces a program that evaluates to an equivalent value (modulo type annotations). Now apparently the coherence issue of the declarative system breaks the dynamic guarantee. For instance:

$$(\lambda f : \forall a.a \to a.\ \lambda x : \mathsf{Int}.\ f\ x)\ (\lambda x.\ x)\ 3 \qquad (\lambda f : \forall a.a \to a.\ \lambda x : \star.\ f\ x)\ (\lambda x.\ x)\ 3$$

The left one evaluates to 3, whereas its less precise version (right) will give a cast error if a is instantiated to Bool for example.

As discussed in Sect. 6.2, we could design a more sophisticated declarative/algorithmic type system where coherence is retained. However, even with a coherent source language, the dynamic guarantee is still a question. Currently, the dynamic guarantee for our target language $\lambda \mathsf{B}$ is still an open question. According to Igarashi et al. [14], the difficulty lies in the definition of term precision that preserves the semantics.

7 Related Work

Along the way we discussed some of the most relevant work to motivate, compare and promote our gradual typing design. In what follows, we briefly discuss related work on gradual typing and polymorphism.

Gradual Typing. The seminal paper by Siek and Taha [21] is the first to propose gradual typing. The original proposal extends the simply typed lambda calculus by introducing the unknown type $\star$ and replacing type equality with type consistency. Later Siek and Taha [22] incorporated gradual typing into a simple object oriented language, and showed that subtyping and consistency are orthogonal – an insight that partly inspired our work. We show that subtyping and consistency are orthogonal in a much richer type system with higher-rank polymorphism. Siek et al. [25] proposed a set of criteria that provides important guidelines for designers of gradually typed languages. Cimini and Siek [8] introduced the *Gradualizer*, a general methodology for generating gradual type systems from static type systems. Later they also develop an algorithm to generate dynamic semantics [9]. Garcia et al. [13] introduced the AGT approach based on abstract interpretation.

Gradual Type Systems with Explicit Polymorphism. Ahmed et al. [1] proposed $\lambda\mathsf{B}$ that extends the blame calculus [29] to incorporate polymorphism. The key novelty of their work is to use dynamic sealing to enforce parametricity. Devriese et al. [10] proved that embedding of System F terms into $\lambda\mathsf{B}$ is not fully abstract. Igarashi et al. [14] also studied integrating gradual typing with parametric polymorphism. They proposed System F_G, a gradually typed extension of System F, and System F_C, a new polymorphic blame calculus. As has been discussed extensively, their definition of type consistency does not apply to our setting (implicit polymorphism). All of these approaches mix consistency with subtyping to some extent, which we argue should be orthogonal.

Gradual Type Inference. Siek and Vachharajani [23] studied unification-based type inference for gradual typing, where they show why three straightforward approaches fail to meet their design goals. Their type system infers gradual types, which results in a complicated type system and inference algorithm. Garcia and Cimini [12] presented a new approach where gradual type inference only produces static types, which is adopted in our type system. They also deal with let-polymorphism (rank 1 types). However none of these works deals with higher-ranked implicit polymorphism.

Higher-Rank Implicit Polymorphism. Odersky and Läufer [17] introduced a type system for higher-rank types. Based on that, Peyton Jones et al. [18] developed an approach for type checking higher-rank predicative polymorphism. Dunfield and Krishnaswami [11] proposed a bidirectional account of higher-rank polymorphism, and an algorithm for implementing the declarative system, which serves as a sole inspiration for our algorithmic system. The key difference, however, is the integration of gradual typing. Vytiniotis et al. [28] defers static type errors to runtime, which is fundamentally different from gradual typing, where programmers can control over static or runtime checks by precision of the annotations.

8 Conclusion

In this paper, we present a generalized definition of consistent subtyping, which is proved to be applicable to both polymorphic and top types. Based on the new definition of consistent subtyping, we have developed a gradually typed calculus with predicative implicit higher-rank polymorphism, and an algorithm to implement it. As future work, we are interested to investigate if our results can scale to real world languages and other programming language features.

Acknowledgements. We thank Ronald Garcia and the anonymous reviewers for their helpful comments. This work has been sponsored by the Hong Kong Research Grant Council projects number 17210617 and 17258816.

References

1. Ahmed, A., Findler, R.B., Siek, J.G., Wadler, P.: Blame for all. In: Proceedings of the 38th Symposium on Principles of Programming Languages (2011)
2. Ahmed, A., Jamner, D., Siek, J.G., Wadler, P.: Theorems for free for free: parametricity, with and without types. In: Proceedings of the 22nd International Conference on Functional Programming (2017)
3. Schwerter, F.B., Garcia, R., Tanter, É.: A theory of gradual effect systems. In: Proceedings of the 19th International Conference on Functional Programming (2014)
4. Bierman, G., Meijer, E., Torgersen, M.: Adding dynamic types to C$^\sharp$. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 76–100. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_5
5. Bierman, G., Abadi, M., Torgersen, M.: Understanding TypeScript. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 257–281. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44202-9_11
6. Bonnaire-Sergeant, A., Davies, R., Tobin-Hochstadt, S.: Practical optional types for clojure. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 68–94. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_4
7. Castagna, G., Lanvin, V.: Gradual typing with union and intersection types. Proc. ACM Program. Lang. **1**(ICFP), 41:1–41:28 (2017)
8. Cimini, M., Siek, J.G.: The gradualizer: a methodology and algorithm for generating gradual type systems. In: Proceedings of the 43rd Symposium on Principles of Programming Languages (2016)
9. Cimini, M., Siek, J.G.: Automatically generating the dynamic semantics of gradually typed languages. In: Proceedings of the 44th Symposium on Principles of Programming Languages (2017)
10. Devriese, D., Patrignani, M., Piessens, F.: Parametricity versus the universal type. Proc. ACM Program. Lang. **2**(POPL), 38 (2017)
11. Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. In: International Conference on Functional Programming (2013)
12. Garcia, R., Cimini, M.: Principal type schemes for gradual programs. In: Proceedings of the 42nd Symposium on Principles of Programming Languages (2015)
13. Garcia, R., Clark, A.M., Tanter, É.: Abstracting gradual typing. In: Proceedings of the 43rd Symposium on Principles of Programming Languages (2016)

14. Igarashi, Y., Sekiyama, T., Igarashi, A.: On polymorphic gradual typing. In: Proceedings of the 22nd International Conference on Functional Programming (2017)
15. Jafery, K.A., Dunfield, J.: Sums of uncertainty: refinements go gradual. In: Proceedings of the 44th Symposium on Principles of Programming Languages (2017)
16. Mitchell, J.C.: Polymorphic type inference and containment. In: Logical Foundations of Functional Programming (1990)
17. Odersky, M., Läufer, K.: Putting type annotations to work. In: Proceedings of the 23rd Symposium on Principles of Programming Languages (1996)
18. Jones, S.P., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. J. Funct. Program. **17**(1), 1–82 (2007)
19. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: Proceedings of the IFIP 9th World Computer Congress (1983)
20. Reynolds, J.C.: The coherence of languages with intersection types. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 675–700. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-54415-1_70
21. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Proceedings of the 2006 Scheme and Functional Programming Workshop (2006)
22. Siek, J., Taha, W.: Gradual typing for objects. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 2–27. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73589-2_2
23. Siek, J.G., Vachharajani, M.: Gradual typing with unification-based inference. In: Proceedings of the 2008 Symposium on Dynamic Languages (2008)
24. Siek, J.G., Wadler, P.: The key to blame: gradual typing meets cryptography (draft) (2016)
25. Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined criteria for gradual typing. In: LIPIcs-Leibniz International Proceedings in Informatics (2015)
26. Verlaguet, J.: Facebook: analyzing PHP statically. In: Proceedings of Commercial Users of Functional Programming (2013)
27. Vitousek, M.M., Kent, A.M., Siek, J.G., Baker, J.: Design and evaluation of gradual typing for Python. In: Proceedings of the 10th Symposium on Dynamic Languages (2014)
28. Vytiniotis, D., Jones, S.P., Magalhães, J.P.: Equality proofs and deferred type errors: a compiler pearl. In: Proceedings of the 17th International Conference on Functional Programming, ICFP 2012, New York (2012)
29. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 1–16. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00590-9_1

HOBiT: Programming Lenses Without Using Lens Combinators

Kazutaka Matsuda[1(✉)] and Meng Wang[2]

[1] Tohoku University, Sendai 980-8579, Japan
kztk@ecei.tohoku.ac.jp
[2] University of Bristol, Bristol BS8 1TH, UK

Abstract. We propose HOBiT, a higher-order bidirectional programming language, in which users can write bidirectional programs in the familiar style of conventional functional programming, while enjoying the full expressiveness of lenses. A bidirectional transformation, or a lens, is a pair of mappings between source and view data objects, one in each direction. When the view is modified, the source is updated accordingly with respect to some laws—a pattern that is found in databases, model-driven development, compiler construction, and so on. The most common way of programming lenses is with lens combinators, which are lens-to-lens functions that compose simpler lenses to form more complex ones. Lens combinators preserve the bidirectionality of lenses and are expressive; but they compel programmers to a specialised point-free style—i.e., no naming of intermediate computation results—limiting the scalability of bidirectional programming. To address this issue, we propose a new bidirectional programming language HOBiT, in which lenses are represented as standard functions, and combinators are mapped to language constructs with binders. This design transforms bidirectional programming, enabling programmers to write bidirectional programs in a flexible functional style and at the same time access the full expressiveness of lenses. We formally define the syntax, type system, and the semantics of the language, and then show that programs in HOBiT satisfy bidirectionality. Additionally, we demonstrate HOBiT's programmability with examples.

1 Introduction

Transforming data from one format to another is a common task of programming: compilers transform program texts into syntax trees, manipulate the trees and then generate low-level code; database queries transform base relations into views; model transformations generate lower-level implementations from higher-level models; and so on. Very often, such transformations will benefit from being bidirectional, allowing changes to the targets to be mapped back to the sources too. For example, if one can run a compiler front-end (preprocessing, parsing, desugaring, etc.) backwards, then all sorts of program analysis tools will be able to focus on a much smaller core language, without sacrificing usability, as

© The Author(s) 2018
A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 31–59, 2018.
https://doi.org/10.1007/978-3-319-89884-1_2

their outputs in term of the core language will be transformed backwards to the source language. In the same way, such needs arise in databases (the *view-update problem* [1,6,12]) and model-driven engineering (bidirectional model transformation) [28,33,35].

As a response to this challenge, programming language researchers have started to design languages that execute deterministically in both directions, and the lens framework is the most prominent among all. In the lens framework, a *bidirectional transformation* (or a *lens*) $\ell \in \mathit{Lens}\ S\ V$, consists of $\mathit{get}\ \ell \in S \to V$, and $\mathit{put}\ \ell \in S \to V \to S$ [3,7,8]. (When clear from the context, or unimportant, we sometimes omit the lens name and write simply *get*/*put*.) Function *get* extracts a view from a source, and *put* takes both an updated view and the original source as inputs to produce an updated source. The additional parameter of *put* makes it possible to recover some of the source data that is not present in the view. In other words, *get* needs not to be injective to have a *put*. Not all pairs of *get*/*put* are considered correct lenses. The following round-triping laws of a lens ℓ are generally required to establish bidirectionality:

$$\mathit{put}\ \ell\ s\ v = s \quad \text{if} \quad \mathit{get}\ \ell\ s = v \qquad (\textbf{Acceptability})$$
$$\mathit{get}\ \ell\ s' = v \quad \text{if} \quad \mathit{put}\ \ell\ s\ v = s' \qquad (\textbf{Consistency})$$

for all s, s' and v. (In this paper we write $e = e'$ with the assumption that neither e nor e' is undefined. Stronger variants of the laws enforcing totality exist elsewhere, for example in [7].) Here *consistency* ensures that all updates on a view are captured by the updated source, and *acceptability* prohibits changes to the source if no update has been made on the view. Collectively, the two laws defines *well-behavedness* [1,7,12].

The most common way of programming lenses is with lens combinators [3,7,8], which are basically a selection of lens-to-lens functions that compose simpler lenses to form more complex ones. This combinator-based approach follows the long history of lightweight language development in functional programming. The distinctive advantage of this approach is that by restricting the lens language to a few selected combinators, well-behavedness can be more easily preserved in programming, and therefore given well-behaved lenses as inputs, the combinators are guaranteed to produce well-behaved lenses. This idea of lens combinators is very influential academically, and various designs and implementations have been proposed [2,3,7–9,16,17,27,32] over the years.

1.1 The Challenge of Programmability

The complexity of a piece of software can be classified as either intrinsic or accidental. Intrinsic complexity reflects the inherent difficulty of the problem at hand, whereas accidental complexity arises from the particular programming language, design or tools used to implement the solution. This work aims at reducing the accidental complexity of bidirectional programming by contributing to the design of bidirectional languages. In particularly, we identify a language restriction—i.e., no naming of intermediate computation results—which complicates lens programming, and propose a new design that removes it.

As a teaser to demonstrate the problem, let us consider the list append function. In standard unidirectional programming, it can be defined simply as $append\ x\ y = \mathbf{case}\ x\ \mathbf{of}\ \{[\,] \to y;\ a : x' \to a : append\ x'\ y\}$. Astute readers may have already noticed that *append* is defined by structural recursion on x, which can be made explicit by using *foldr* as in $append\ x\ y = foldr\ (:)\ y\ x$.

But in a lens language based on combinators, things are more difficult. Specifically, *append* now requires a more complicated recursion pattern, as below.

```
appendL :: Lens ([A], [A]) [A]
appendL =
  cond idL (λ_.True) (λ_.λ_.[ ]) (consL ô (idL × appendL)) (not ∘ null) (λ_.λ_.⊥)
  ô rearr ô (outListL × idL)
 where outListL :: Lens [A] (Either ( ) (A, [A]))
       rearr    :: Lens (Either ( ) (a, b), c) (Either c (a, (b, c)))
       (ô)      :: Lens b c → Lens a b → Lens a c
       cond     :: Lens a c → ... → Lens b c → ... → Lens (Either a b) c
       ...
```

It is beyond the scope of this paper to explain how exactly the definition of *appendL* works, as its obscurity is what this work aims to remove. Instead, we informally describe its behaviour and the various components of the code. The above code defines a lens: forwards, it behaves as the standard *append*, and backwards, it splits the updated view list, and when the length of the list changes, this definition implements (with the grayed part) the bias of keeping the length of the first source list whenever possible (to disambiguate multiple candidate source changes). Here, *cond*, (ô), etc. are lens combinators and *outListL* and *rearr* are auxiliary lenses, as can be seen from their types. Unlike its unidirectional counterpart, *appendL* can no longer be defined as a structural recursion on list; instead it traverses a pair of lists with rather complex rearrangement *rearr*.

Intuitively, the additional grayed parts is intrinsic complexity, as they are needed for directing backwards execution. However, the complicated recursion scheme, which is a direct result of the underlying limitation of lens languages, is certainly accidental. Recall that in the definition of *append*, we were able to use the variable y, which is bound outside of the recursion pattern, inside the body of *foldr*. But the same is not possible with lens combinators which are strictly 'pointfree'. Moreover, even if one could name such variables (points), their usage with lens combinators will be very restricted in order to guarantee well-behavedness [21,23]. This problem is specific to opaque non-function objects such as lenses, and goes well beyond the traditional issues associated with the pointfree programming style.

In this paper, we design a new bidirectional language HOBiT, which aims to remove much of the accidental difficulty found in combinator-based lens programming, and reduces the gap between bidirectional programming and standard functional programming. For example, the following definition in HOBiT implements the same lens as *appendL*.

$$
\begin{array}{l}
appendB :: \mathbf{B}[A] \to \mathbf{B}[A] \to \mathbf{B}[A] \\
appendB\ x\ y = \underline{\mathbf{case}}\ x\ \underline{\mathbf{of}}\ [\,] \quad \to y \qquad \underline{\mathbf{with}}\ \lambda\_.\mathsf{True} \quad \underline{\mathbf{by}}\ (\lambda\_.\lambda\_.[\,]) \\
\qquad\qquad\qquad\qquad\quad a : x' \to a \mathbin{\underline{:}} appendB\ x'\ y\ \underline{\mathbf{with}}\ not \circ null\ \underline{\mathbf{by}}\ (\lambda\_.\lambda\_.\bot)
\end{array}
$$

As expected, the above code shares the grayed part with the definition of *appendL* as the two implement the same backwards behaviour. The difference is that *appendB* uses structural recursion in the same way as the standard unidirectional *append*, greatly simplifying programming. This is made possible by the HOBiT's type system and semantics, allowing unrestricted use of free variables. This difference in approach is also reflected in the types: *appendB* is a proper function (instead of the abstract lens type of *appendL*), which readily lends itself to conventional functional programming. At the same time, *appendB* is also a proper lens, which when executed by the HOBiT interpreter behave exactly like *appendL*. A major technical challenge in the design of HOBiT is to guarantee this duality, so that functions like *appendB* are well-behaved by construction despite the flexibility in their construction.

1.2 Contributions

As we can already see from the very simple example above, the use of HOBiT simplifies bidirectional programming by removing much of the accidental complexity. Specifically, HOBiT stands out from existing bidirectional languages in two ways:

1. It supports the conventional programming style that is used in unidirectional programming. As a result, a program in HOBiT can be defined in a way similar to how one would define only its *get* component. For example, *appendB* is defined in the same way as the unidirectional *append*.
2. It supports incremental improvement. Given the very often close resemblance of a bidirectional-program definition and that of its *get* component, it becomes possible to write an initial version of a bidirectional program almost identical to its *get* component and then to adjust the backwards behaviour gradually, without having to significantly restructure the existing definition.

Thanks to these distinctive advantages, HOBiT for the first time allows us to construct realistically-sized bidirectional programs with relative ease. Of course, this does not mean free lunch: the ability to control backwards behaviours will not magically come without additional code (for example the grayed part above). What HOBiT achieves is that programming effort may now focus on the productive part of specifying backwards behaviours, instead of being consumed by circumventing language restrictions.

In summary, we make the following contributions in this paper.

– We design a higher-order bidirectional programming language HOBiT, which supports convenient bidirectional programming with control of backwards behaviours (Sect. 3). We also discuss several extensions to the language (Sect. 5).

- We present the semantics of HOBiT inspired by the idea of staging [5], and prove the well-behavedness property using Kripke logical relations [18] (Sect. 4).
- We demonstrate the programmability of HOBiT with examples such as desugaring/resugaring [26] (Sect. 6). Additional examples including a bidirectional evaluator for λ-calculus [21,23], a parser/printer for S-expressions, and bookmark extraction for Netscape [7] can be found at https://bitbucket.org/kztk/hibx together with a prototype implementation of HOBiT.

2 Overview: Bidirectional Programming Without Combinators

In this section, we informally introduce the essential constructs of HOBiT and demonstrate their use by a few small examples. Recall that, as seen in the *appendB* example, the strength of HOBiT lies in allowing programmers to access λ-abstractions without restrictions on the use of λ-bound variables.

2.1 The $\underline{\textbf{case}}$ Construct

The most important language construct in HOBiT is $\underline{\textbf{case}}$ (pronounced as *bidirectional case*), which provides pattern matching and easy access to bidirectional branching, and also importantly, allows unrestricted use of λ-bound variables.

In general, a $\underline{\textbf{case}}$ expression has the following form.

$$\underline{\textbf{case}}\ e\ \underline{\textbf{of}}\ \{p_1 \to e_1\ \underline{\textbf{with}}\ \phi_1\ \underline{\textbf{by}}\ \rho_1; \ldots; p_n \to e_n\ \underline{\textbf{with}}\ \phi_n\ \underline{\textbf{by}}\ \rho_n\}$$

(Like Haskell, we shall omit "{", "}" and ";" if they are clear from the layout.) In the type system of HOBiT, a $\underline{\textbf{case}}$-expression has type $\mathbf{B}B$, if e and e_i have types $\mathbf{B}A$ and $\mathbf{B}B$, and ϕ_i and ρ_i have types $B \to \mathit{Bool}$ and $A \to B \to A$, where A and B contains neither $(\to)$ nor $\mathbf{B}$. The type $\mathbf{B}A$ can be understood intuitively as "updatable A". Typically, the source and view data are given such $\mathbf{B}$-types, and a function of type $\mathbf{B}A \to \mathbf{B}B$ is the HOBiT equivalent of *Lens A B*.

The pattern matching part of $\underline{\textbf{case}}$ performs two implicit operations: it first unwraps the $\mathbf{B}$-typed value, exposing its content for normal pattern matching, and then it wraps the variables bound by the pattern matching, turning them into 'updatable' $\mathbf{B}$-typed values to be used in the bodies. For example, in the second branch of *appendB*, a and x' can be seen as having types A and $[A]$ in the pattern, but $\mathbf{B}A$ and $\mathbf{B}[A]$ types in the body; and the bidirectional constructor $(\underline{:}) :: \mathbf{B}A \to \mathbf{B}[A] \to \mathbf{B}[A]$ combines them to produce a $\mathbf{B}$-typed list.

In addition to the standard conditional branches, $\underline{\textbf{case}}$-expression has two unique components ϕ_i and ρ_i called *exit conditions* and *reconciliation functions* respectively, which are used in backwards executions. Exit condition ϕ_i is an over-approximation of the forwards-execution results of the expressions e_i. In other words, if branch i is choosen, then $\phi_i\ e_i$ must evaluate to True. This assertion is checked dynamically in HOBiT, though could be checked statically with a sophisticated type system [7]. In the backwards direction the exit condition is used for deciding branching: the branch with its exit condition satisfied by the updated view (when more than one match, the original branch used in the forwards direction has higher priority) will be picked for execution. The idea is that due to the update in the view, the branch taken in the backwards direction may be different from the one taken in the original forwards execution, a feature that is commonly supported by lens languages [7] which we call *branch switching*.

Branch switching is crucial to *put*'s *robustness*, i.e., the ability to handle a wide range of view updates (including those affect the branching decisions) without failing. We explain its working in details in the following.

Branch Switching. Being able to choose a different branch in the backwards direction only solves part of the problem. Let us consider the case where a forward execution chooses the n^{th} branch, and the backwards execution, based on the updated view, chooses the m^{th} $(m \neq n)$ branch. In this case, the original value of the pattern-matched expression e, which is the reason for the n^{th} branch being chosen, is not compatible with the *put* of the m^{th} branch.

As an example, let us consider a simple function that pattern-matches on an *Either* structure and returns an list. Note that we have purposely omitted the reconciliation functions.

$$
\begin{array}{l}
f :: \mathbf{B}(\mathit{Either}\ [A]\ (A,[A])) \to \mathbf{B}[A] \\
f\ x = \underline{\mathbf{case}}\ x\ \underline{\mathbf{of}}\ \begin{array}[t]{llll}\mathsf{Left}\ \mathit{ys} & \to \mathit{ys} & \underline{\mathbf{with}}\ \lambda\_.\mathsf{True} & \text{\{- no } \underline{\mathbf{by}} \text{ here -\}} \\ \mathsf{Right}\ (y,\mathit{ys}) & \to y\ \underline{:}\ \mathit{ys} & \underline{\mathbf{with}}\ \mathit{not} \circ \mathit{null} & \end{array}
\end{array}
$$

We have said that functions of type $\mathbf{B}A \to \mathbf{B}B$ are also fully functioning lenses of type *Lens A B*. In HOBiT, the above code runs as follows, where `HOBiT>` is the prompt of HOBiT's read-eval-print loop, and `:get` and `:put` are meta-language operations to perform *get* and *put* respectively.

```
HOBiT> :get f (Left [1,2,3])
[1,2,3]
HOBiT> :get f (Right (1,[2,3]))
[1,2,3]
HOBiT> :put f (Left [1,2,3]) [4,5]      -- The view [1,2,3] is updated to [4,5].
Left [4,5]                              -- Both exit conditions are true with [4,5],
                                        -- so the original branch (Left) is taken.
HOBiT> :put f (Right (1,[2,3])) [4,5]
Right (4,[5])                           -- Similar, but the original branch is Right.
HOBiT> :put f (Right (1,[2,3])) []
⊥                                       -- Branch switches, but computation fails.
```

As we have explained above, exit conditions are used to decide which branch will be used in the backwards direction. For the first and second evaluations of *put*, the exit conditions corresponding to the original branches were true for the updated view. For the last evaluation of *put*, since the exit condition of

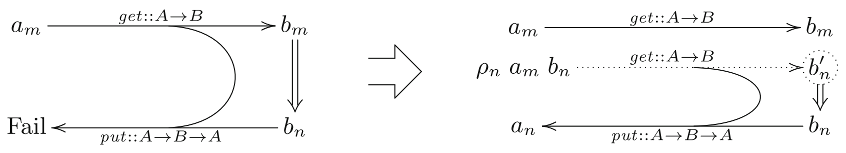


Fig. 1. Reconciliation function: assuming exit conditions ϕ_m and ϕ_n where $\phi_m\ b_n = \mathsf{False}$ but $\phi_n\ b_n = \mathsf{True}$, and reconciliation functions ρ_m and ρ_n.

the original branch was false but that of the other branch was true, branch switching is required here. However, a direct *put*-execution of f with the inputs $(\mathsf{Right}\ (1, [2, 3]))$ and $[\,]$ crashes (represented by $\bot$ above), for a good reason, as the two inputs are in an inconsistent state with respect to f.

This is where reconciliation functions come into the picture. For the Left branch above, a sensible reconciliation function will be $(\lambda\_.\lambda\_.\mathsf{Left}\ [\,])$, which when applied turns the conflicting source $(\mathsf{Right}\ (1, [2, 3]))$ into $\mathsf{Left}\ [\,]$, and consequently the *put*-execution may succeed with the new inputs and returns $\mathsf{Left}\ [\,]$. It is not difficult to verify that the "reconciled" *put*-execution still satisfies well-behavedness. Note that despite the similarity in types, reconciliation functions are not *put*; they merely provide a default source value to allow stuck *put*-executions to proceed. We visualise the effect of reconciliation functions in Fig. 1. The left-hand side is bidirectional execution without successful branch-switching, and since $\phi_m\ b_n$ is false (indicating that b_n is not in the range of the m^{th} branch) the execution of *put* must (rightfully) fail in order to guarantee well-behavedness. On the right-hand side, reconciliation function ρ_n produces a suitable source from a_m and b_n (where $\phi_n\ (get\ (\rho_n\ a_m\ b_n))$ is True), and *put* executes with b_n and the new source $\rho_n\ a_m\ b_n$. It is worth mentioning that branch switching with reconciliation functions does not compromise correctness: though the quality of the user-defined reconciliation functions affects robustness as they may or may not be able to resolve conflicts, successful *put*-executions always guarantee well-behavedness, regardless the involvement of reconciliation functions.

Revisiting *appendB*. Recall *appendB* from Sect. 1.1 (reproduced below).

```
appendB :: B[A] → B[A] → B[A]
appendB x y = case x of []      → y                    with λ_.True   by (λ_.λ_.[])
                        a : x′ → a : appendB x′ y with not ∘ null by (λ_.λ_.⊥)
```

The exit condition for the nil case always returns true as there is no restriction on the value of y, and for the cons case it requires the returned list to be non-empty. In the backwards direction, when the updated view is non-empty, both exit conditions will be true, and then the original branch will be taken. This means that since *appendB* is defined as a recursion on x, the backwards execution will try to unroll the original recursion step by step (i.e., the cons branch will be taken for a number of times that is the same as the length of x) as long as the view remains non-empty. If an updated view list is shorter than x, then $not \circ null$

will become false before the unrolling finishes, and the nil branch will be taken (branch-switching) and the reconciliation function will be called.

The definition of *appendB* is curried; straightforward uncurrying turns it into the standard form $\mathbf{B}A \to \mathbf{B}B$ that can be interpreted by HOBiT as a lens. The following HOBiT program is the bidirectional variant of *uncurry*.

$$\begin{aligned}&\mathit{uncurryB} :: (\mathbf{B}A \to \mathbf{B}B \to \mathbf{B}C) \to \mathbf{B}(A, B) \to \mathbf{B}C\\&\mathit{uncurryB}\ f\ z = \underline{\mathbf{let}}\ (x, y) = z\ \underline{\mathbf{in}}\ f\ x\ y\end{aligned}$$

Here, $\underline{\mathbf{let}}\ p = e\ \underline{\mathbf{in}}\ e'$ is syntactic sugar for $\underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \{p \to e'\ \underline{\mathbf{with}}\ (\lambda\_.\mathsf{True})\ \underline{\mathbf{by}}\ (\lambda s.\lambda\_.s)\}$, in which the reconciliation function is never called as there is only one branch. Let $\mathit{appendB}' = \mathit{uncurryB}\ \mathit{appendB}$, then we can run $\mathit{appendB}'$ as:

```
HOBiT> :get appendB' ([1,2],[3,4,5])
[1,2,3,4,5]
HOBiT> :put appendB' ([1,2],[3,4,5]) [6,7,8,9,10]
([6,7],[8,9,10])       -- No structural change, no branch switching.
HOBiT> :put appendB' ([1,2],[3,4,5]) [6,7]
([6,7],[])       -- No branch switching, still.
HOBiT> :put appendB' ([1,2],[3,4,5]) [6]
([6],[])       -- Branch-switching happens and the recursion terminates early.
```

Difference from Lens Combinators. As mentioned above, the idea of branch switching can be traced back to lens languages. In particular, the design of **case** is inspired by the combinator *cond* [7]. Despite the similarities, it is important to recognise that **case** is not only a more convenient syntax for *cond*, but also crucially supports the unrestricted use of λ-bound variables. This more fundamental difference is the reason why we could define *appendB* in the conventional functional style as the variables x and y are used freely in the body of **case**. In other words, the novelty of HOBiT is its ability to combine the traditional (higher-order) functional programming and the bidirectional constructs as found in lens combinators, effectively establishing a new way of bidirectional programming.

2.2 A More Elaborate Example: *linesB*

In addition to supporting convenient programming and robustness in *put* execution, the **case** constructs can also be used to express intricate details of backwards behaviours. Let us consider the *lines* function in Haskell as an example, which splits a string into a list of strings by newlines, for example, *lines* `"AA\nBB\n"` = [`"AA"`, `"BB"`], except that the last newline character in its input is optional. For example, *lines* returns [`"AA"`, `"BB"`] for both `"AA\nBB\n"` and `"AA\nBB"`. Suppose that we want the backwards transformation of *lines* to exhibit a behaviour that depends on the original source:

```
linesB :: BString → B[String]
linesB str =
   let (f, b) = breakNLB str
   in case b of '\n' : x : r → f : linesB (x : r)
                                        with (> 1) ∘ length by (λb.λ_.'\n' : ' ' : b)
                 b'          → f : [] with (== 1) ∘ length by (λb.λ_.lastNL b)
  where {lastNL [] = []; lastNL ['\n'] = ['\n']; lastNL (a : x) = lastNL x}
breakNLB :: BString → B(String, String)
breakNLB str = case str of
   []       → ([], [])                                 with p1 by (λ_.λ_.[])
   '\n' : s → ([], '\n' : s)                           with p2 by (λ_.λ_."\n")
   c : s    → let (f, r) = breakNLB s in (c : f, r)    with p3 by (λ_.λ_." ")
  where {p1(x, y) = null y;  p2(x, y) = null x && not (null y);  p3(x, y) = not (null x)}
```

Fig. 2. *linesB* and *breakNLB*

```
HOBiT> :put linesB "AA\nBB" ["a", "b"]
"a\nb"
HOBiT> :put linesB "AA\nBB" ["a", "b", "c"]
"a\nb\nc"
HOBiT> :put linesB "AA\nBB" ["a"]
"a"
HOBiT> :put linesB "AA\nBB\n" ["a", "b", "c"]
"a\nb\nc\n"
HOBiT> :put linesB "AA\nBB\n" ["a"]
"a\n"
```

This behaviour is achieved by the definition in Fig. 2, which makes good use of reconciliation functions. Note that we do not consider the contrived corner case where the string ends with duplicated newlines such as in `"A\n\n"`. The function *breakNLB* splits a string at the first newline; since *breakNLB* is injective, its exit conditions and reconciliation functions are of little interest. The interesting part is in the definition of *linesB*, particularly its use of reconciliation functions to track the existence of a last newline character. We firstly explain the branching structure of the program. On the top level, when the first line is removed from the input, the remaining string b may contain more lines, or be the end (represented by either the empty list or the singleton list `['\n']`). If the first branch is taken, the returned result will be a list of more than one element. In the second branch when it is the end of the text, b could contain a newline or simply be empty. We do not explicitly give patterns for the two cases as they have the same body $f \,\underline{:}\, \underline{[\,]}$, but the reconciliation function distinguishes the two in order to preserve the original source structure in the backwards execution. Note that we intentionally use the same variable name b in the case analysis and the reconciliation function, to signify that the two represent the same source data. The use of argument b in the reconciliation functions serves the purpose of remembering the (non)existence of the last newline in the original source, which is then preserved in the new source.

$$\begin{aligned} e &::= x \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{True} \mid \mathsf{False} \mid [\,] \mid e_1 : e_2 \mid \mathbf{case}\ e\ \mathbf{of}\ \{p_i \to e_i\}_{i=1,2} \mid \mathbf{fix}\ (\lambda f.e) \\ &\quad\mid\ \underline{\mathsf{True}} \mid \underline{\mathsf{False}} \mid \underline{[\,]} \mid e_1 \mathbin{\underline{:}} e_2 \mid \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ e_i'\ \underline{\mathbf{by}}\ e_i''\}_{i=1,2} \\ p &::= x \mid \mathsf{True} \mid \mathsf{False} \mid [\,] \mid p_1 : p_2 \end{aligned}$$

Fig. 3. Syntax of HOBiT Core

It is worth noting that just like the other examples we have seen, this definition in HOBiT shares a similar structure with a definition of *lines* in Haskell.[1] The notable difference is that a Haskell definition is likely to have a different grouping of the three cases of *lines* into two branches, as there is no need to keep track of the last newline for backwards execution. Recall that reconciliation functions are called *after* branches are chosen by exit conditions; in the case of *linesB*, the reconciliation function is used to decide the reconciled value of b' to be `"\n"` or `""`. This, however, means that we cannot separate the pattern b' into two `"\n"` and `""` with copying its branch body and exit condition, because then we lose a chance to choose a reconciled value of b based on its original value.

3 Syntax and Type System of HOBiT Core

In this section, we describe the syntax and the type system of the core of HOBiT.

3.1 Syntax

The syntax of HOBiT Core is given in Fig. 3. For simplicity, we only consider booleans and lists. The syntax is almost the same as the standard λ-calculus with the fixed-point combinator (**fix**), lists and booleans. For data constructors and case expressions, there are in addition bidirectional versions that are underlined. We allow the body of **fix** to be non-λs to make our semantics simple (Sect. 4), though such a definition like $\mathbf{fix}(\lambda x.\mathsf{True} : x)$ can diverge.

Although in examples we used **case**/$\underline{\mathbf{case}}$-expressions with an arbitrary number of branches having overlapping patterns under the first-match principle, we assume for simplicity that in HOBiT Core **case**/$\underline{\mathbf{case}}$-expressions must have exactly two branches whose patterns do not overlap; extensions to support these features are straightforward. As in Haskell, we sometimes omit the braces and semicolons if they are clear from the layout.

[1] Haskell's *lines*'s behaviour is a bit more complicated as it returns $[\,]$ if and only if the input is `""`. This behaviour can be achieved by calling *linesB* only when the input list is nonempty.

$$\boxed{\Gamma; \Delta \vdash e : A}$$

$$\frac{\Gamma(x) = A}{\Gamma;\Delta \vdash x : A} \qquad \frac{\Delta(x)=\sigma}{\Gamma;\Delta\vdash x : \mathbf{B}\sigma} \qquad \frac{\Gamma, x:A;\Delta\vdash e:B}{\Gamma;\Delta\vdash \lambda x.e : A\to B} \qquad \frac{\Gamma;\Delta\vdash e_1 : A\to B \quad \Gamma;\Delta\vdash e_2 : A}{\Gamma;\Delta\vdash e_1\ e_2 : B}$$

$$\frac{\Gamma, f : A;\Delta\vdash e : A}{\Gamma;\Delta\vdash \mathbf{fix}(\lambda f.e) : A} \qquad \frac{}{\Gamma;\Delta\vdash \mathsf{True} : \mathit{Bool}} \qquad \frac{}{\Gamma;\Delta\vdash \mathsf{False} : \mathit{Bool}} \qquad \frac{}{\Gamma;\Delta\vdash [\,] : [A]}$$

$$\frac{\Gamma;\Delta\vdash e_1 : A \quad \Gamma;\Delta\vdash e_2 : [A]}{\Gamma;\Delta\vdash e_1 : e_2 : [A]} \qquad \frac{}{\Gamma;\Delta\vdash \underline{\mathsf{True}} : \mathbf{B}\mathit{Bool}} \qquad \frac{}{\Gamma;\Delta\vdash \underline{\mathsf{False}} : \mathbf{B}\mathit{Bool}} \qquad \frac{}{\Gamma;\Delta\vdash \underline{[\,]} : \mathbf{B}[\sigma]}$$

$$\frac{\Gamma;\Delta\vdash e_1 : \mathbf{B}\sigma \quad \Gamma;\Delta\vdash e_2 : \mathbf{B}[\sigma]}{\Gamma;\Delta\vdash e_1 \underline{:} e_2 : \mathbf{B}[\sigma]} \qquad \frac{\Gamma;\Delta\vdash e : A \quad \Gamma_i\vdash p_i : A \quad \Gamma,\Gamma_i;\Delta\vdash e_i : B \quad (i=1,2)}{\Gamma;\Delta\vdash \mathbf{case}\ e\ \mathbf{of}\ \{p_i\to e_i\}_{i=1,2} : B}$$

$$\frac{\begin{array}{c}\Gamma;\Delta\vdash e : \mathbf{B}\sigma \quad \Delta_i\vdash p_i : \sigma \quad \Gamma;\Delta,\Delta_i\vdash e_i : \mathbf{B}\tau \\ \Gamma;\Delta\vdash e'_i : \tau\to \mathit{Bool} \quad \Gamma;\Delta\vdash e''_i : \sigma\to\tau\to\sigma \quad (i=1,2)\end{array}}{\Gamma;\Delta\vdash \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \{p_i\to e_i\ \underline{\mathbf{with}}\ e'_i\ \underline{\mathbf{by}}\ e''_i\}_{i=1,2} : \mathbf{B}\tau}$$

$$\boxed{\Gamma\vdash p : A}$$

$$\frac{}{x : A\vdash x : A} \qquad \frac{}{\emptyset\vdash \mathsf{True} : \mathit{Bool}} \qquad \frac{}{\emptyset\vdash \mathsf{False} : \mathit{Bool}} \qquad \frac{}{\emptyset\vdash [\,] : [A]} \qquad \frac{\Gamma_1\vdash e_1 : A \quad \Gamma_2\vdash e_2 : [A]}{\Gamma_1,\Gamma_2\vdash e_1 : e_2 : [A]}$$

Fig. 4. Typing rules: $\Delta \vdash p : \sigma$ is similar to $\Gamma \vdash p : A$ but asserts that the resulting environment is actually a bidirectional environment.

3.2 Type System

The types in HOBiT Core are defined as follows.

$$A, B ::= \mathbf{B}\sigma \mid A \to B \mid [A] \mid \mathit{Bool}$$

We use the metavariable $\sigma, \tau, \ldots$ for types that do not contain $\to$ nor $\mathbf{B}$, We call σ-types *pure datatypes*, which are used for sources and views of lenses. Intuitively, $\mathbf{B}\sigma$ represents "updatable σ"—data subject to update in bidirectional transformation. We keep the type system of HOBiT Core simple, though it is possible to include polymorphic types or intersection types to unify unidirectional and bidirectional constructors.

The typing judgment $\Gamma; \Delta \vdash e : A$, which reads that under environments Γ and Δ, expression e has type A, is defined by the typing rules in Fig. 4. We use two environments: Δ (the *bidirectional type environment*) is for variables introduced by pattern-matching through $\underline{\mathbf{case}}$, and Γ for everything else. It is interesting to observe that Δ only holds pure datatypes, as the pattern variables of $\underline{\mathbf{case}}$ have pure datatypes, while Γ holds any types. We assume that the variables in Γ and those in Δ are disjoint, and appropriate α-renaming has been done to ensure this. This separation of Δ from Γ does not affect typeability, but is key to our semantics and correctness proof (Sect. 4). Most of the rules are standard except $\underline{\mathbf{case}}$; recall that we only use unidirectional constructors in patterns which have pure types, while the variables bound in the patterns are used as $\mathbf{B}$-typed values in branch bodies.

4 Semantics of HOBiT Core

Recall that the unique strength of HOBiT is its ability to mix higher-order unidirectional programming with bidirectional programming. A consequence of this mixture is that we can no longer specify its semantics in the same way as other *first-order* bidirectional languages such as [13], where two semantics—one for *get* and the other for *put*—suffice. This is because the category of lenses is believed to have no exponential objects [27] (and thus does not permit λs).

4.1 Basic Idea: Staging

Our solution to this problem is staging [5], which separates evaluation into two stages: the unidirectional parts is evaluated first to make way for a bidirectional semantics, which only has to deal with the residual first-order programs. As a simple example, consider the expression $(\lambda z.z)\ (x \mathbin{\underline{:}} ((\lambda w.w)\ y) \mathbin{\underline{:}} \underline{[\,]})$. The first-stage evaluation, $e \Downarrow_{\mathrm{U}} E$, eliminates λs from the expression as in $(\lambda z.z)\ (x \mathbin{\underline{:}} ((\lambda w.w)\ y) \mathbin{\underline{:}} \underline{[\,]}) \Downarrow_{\mathrm{U}} x \mathbin{\underline{:}} y \mathbin{\underline{:}} \underline{[\,]}$. Then, our bidirectional semantics will be able to treat the residual expression as a lens between value environments and values, following [13,20]. Specifically, we have the *get* evaluation relation $\mu \vdash_{\mathrm{G}} E \Rightarrow v$, which computes the value v of E under environment μ as usual, and the *put* evaluation relation $\mu \vdash_{\mathrm{P}} v \Leftarrow E \dashv \mu'$, which computes an updated environment μ' for E from the updated view v and the original environment μ. In pseudo syntax, it can be understood as *put* $E\ \mu\ v = \mu'$, where μ represents the original source and μ' the new source.

It is worth mentioning that a complete separation of the stages is not possible due to the combination of **fix** and **<u>case</u>**, as an attempt to fully evaluate them in the first stage will result in divergence. Thus, we delay the unidirectional evaluation inside **<u>case</u>** to allow **fix**, and consequently the three evaluation relations (uni-directional, *get*, and *put*) are mutually dependent.

4.2 Three Evaluation Relations: Unidirectional, *get* and *put*

First, we formally define the set of residual expressions:

$$\begin{array}{rl} E ::= & \mathsf{True} \mid \mathsf{False} \mid [\,] \mid E_1 : E_2 \mid \lambda x.e \\ \mid & x \mid \underline{\mathsf{True}} \mid \underline{\mathsf{False}} \mid \underline{[\,]} \mid E_1 \mathbin{\underline{:}} E_2 \mid \underline{\mathbf{case}}\ E_0\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ E_i\ \underline{\mathbf{by}}\ E'_i\}_{i=1,2} \end{array}$$

They are treated as values in the unidirectional evaluation, and as expressions in the *get* and *put* evaluations. Notice that e or e_i appear under λ or **<u>case</u>**, meaning that their evaluations are delayed.

The set of *(first-order) values* is defined as below.

$$v ::= \mathsf{True} \mid \mathsf{False} \mid [\,] \mid v_1 : v_2$$

Accordingly, we define a *(first-order) value environment* μ as a finite mapping from variables to first-order values.

$$\frac{}{x \Downarrow_{\mathrm{U}} x} \qquad \frac{e_1 \Downarrow_{\mathrm{U}} \lambda x.e \quad e_2 \Downarrow_{\mathrm{U}} E_2 \quad e[E_2/x] \Downarrow_{\mathrm{U}} E}{e_1\ e_2 \Downarrow_{\mathrm{U}} E} \qquad \frac{}{\lambda x.e \Downarrow_{\mathrm{U}} \lambda x.e} \qquad \frac{e[\mathbf{fix}(\lambda f.e)/f] \Downarrow_{\mathrm{U}} E}{\mathbf{fix}(\lambda f.e) \Downarrow_{\mathrm{U}} E}$$

$$\frac{e_0 \Downarrow_{\mathrm{U}} E_0 \quad e_i' \Downarrow_{\mathrm{U}} E_i' \quad e_i'' \Downarrow_{\mathrm{U}} E_i'' \quad (i = 1,2)}{\underline{\mathbf{case}}\ e_0\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ e_i'\ \underline{\mathbf{by}}\ e_i''\}_{i=1,2} \Downarrow_{\mathrm{U}} \underline{\mathbf{case}}\ E_0\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ E_i'\ \underline{\mathbf{by}}\ E_i''\}_{i=1,2}}$$

Fig. 5. Evaluation rules for unidirectional parts (excerpt)

Unidirectional Evaluation Relation. The rules for the unidirectional evaluation relation is rather standard, as excerpted in Fig. 5. The bidirectional constructs (i.e., bidirectional constructors and **case**) are frozen, i.e., behave just like ordinary constructors in this evaluation. Notice that we can evaluate an expression containing free variables; then the resulting residual expression may contain the free variables.

Bidirectional (*get* and *put*) Evaluation Relations. The *get* and *put* evaluation relations, $\mu \vdash_{\mathrm{G}} E \Rightarrow v$ and $\mu \vdash_{\mathrm{P}} v \Leftarrow E \dashv \mu'$, are defined so that they together form a lens.

Weakening of Environment. Before we lay out the semantics, it is worth explaining a subtlety in environment handling. In conventional evaluation semantics, a larger than necessary environment does no harm, as long as there is no name clashes. For example, whether the expression x is evaluated under the environment $\{x = 1\}$ or $\{x = 1, y = 2\}$ does not matter. However, the same is not true for bidirectional evaluation. Let us consider a residual expression $E = x \mathbin{\underline{:}} y \mathbin{\underline{:}} \underline{[\,]}$, and a value environment $\mu = \{x = 1, y = 2\}$ as the original source. We expect to have $\mu \vdash_{\mathrm{G}} E \Rightarrow 1 : 2 : [\,]$, which may be derived as:

$$\frac{\dfrac{}{\mu \vdash_{\mathrm{G}} x \Rightarrow 1} \qquad \dfrac{\vdots}{\mu \vdash_{\mathrm{G}} y \mathbin{\underline{:}} \underline{[\,]} \Rightarrow 2 : [\,]}}{\mu \vdash_{\mathrm{G}} x \mathbin{\underline{:}} y \mathbin{\underline{:}} \underline{[\,]} \Rightarrow 1 : 2 : [\,]}$$

In the *put* direction, for an updated view say $3 : 4 : [\,]$, we expect to have $\mu \vdash_{\mathrm{P}} 3 : 4 : [\,] \Leftarrow E \dashv \{x = 3, y = 4\}$ with the corresponding derivation:

$$\frac{\dfrac{}{\mu \vdash_{\mathrm{P}} 3 \Leftarrow x \dashv ?_1} \qquad \dfrac{\vdots}{\mu \vdash_{\mathrm{P}} 4 : [\,] \Leftarrow y \mathbin{\underline{:}} \underline{[\,]} \dashv ?_2}}{\mu \vdash_{\mathrm{P}} 3 : 4 : [\,] \Leftarrow x \mathbin{\underline{:}} y \mathbin{\underline{:}} \underline{[\,]} \dashv \{x = 3, y = 4\}}$$

What shall the environments $?_1$ and $?_2$ be? One way is to have $\mu \vdash_{\mathrm{P}} 3 \Leftarrow x \dashv \{x = 3, y = 2\}$, and $\mu \vdash_{\mathrm{P}} 4 : [\,] \Leftarrow y \mathbin{\underline{:}} \underline{[\,]} \dashv \{x = 1, y = 4\}$, where the variables do not appear free in the residual expression takes their values from the original source environment μ. However, the evaluation will get stuck here, as there is no reasonable way to produce the expected result $\{x = 3, y = 4\}$ from $?_1 = \{x = 3, y = 2\}$ and $?_2 = \{x = 1, y = 4\}$. In other words, the redundancy in environment is harmful as it may cause conflicts downstream.

Our solution to this problem, which follows from [21–23, 29], is to allow *put* to return value environments containing only bindings that are relevant for the residual expressions under evaluation. For example, we have $\mu \vdash_{\mathrm{P}} 3 \Leftarrow x \dashv \{x = 3\}$, and $\mu \vdash_{\mathrm{P}} 4 : [\,] \Leftarrow y \,\underline{:}\, \underline{[\,]} \dashv \{y = 4\}$. Then, we can merge the two value environments $?_1 = \{x = 3\}$ and $?_2 = \{y = 4\}$ to obtain the expected result $\{x = 3, y = 4\}$. As a remark, this seemingly simple solution actually has a non-trivial effect on the reasoning of well-behavedness. We defer a detailed discussion on this to Sect. 4.3.

Now we are ready to define *get* and *put* evaluation rules for each bidirectional constructs. For variables, we just lookup or update environments. Recall that μ is a mapping (i.e., function) from variables to (first-order) values, while we use a record-like notation such as $\{x = v\}$.

$$\frac{}{\mu \vdash_{\mathrm{G}} x \Rightarrow \mu(x)} \qquad \frac{}{\mu \vdash_{\mathrm{P}} v \Leftarrow x \dashv \{x = v\}}$$

For constants $\underline{c}$ where $c = \mathsf{False}, \mathsf{True}, [\,]$, the evaluation rules are straightforward.

$$\frac{}{\mu \vdash_{\mathrm{G}} \underline{c} \Rightarrow c} \qquad \frac{}{\mu \vdash_{\mathrm{P}} c \Leftarrow \underline{c} \dashv \emptyset}$$

The above-mentioned behaviour of the bidirectional cons expression $E_1 \,\underline{:}\, E_2$ is formally given as:

$$\frac{\mu \vdash_{\mathrm{G}} E_1 \Rightarrow v_1 \quad \mu \vdash_{\mathrm{G}} E_2 \Rightarrow v_2}{\mu \vdash_{\mathrm{G}} E_1 \,\underline{:}\, E_2 \Rightarrow v_1 : v_2} \qquad \frac{\mu \vdash_{\mathrm{P}} v_1 \Leftarrow E_1 \dashv \mu_1' \quad \mu \vdash_{\mathrm{P}} v_2 \Leftarrow E_2 \dashv \mu_2'}{\mu \vdash_{\mathrm{P}} v_1 : v_2 \Leftarrow E_1 \,\underline{:}\, E_2 \dashv \mu_1' \curlyvee \mu_2'}$$

(Note that the variable rules guarantee that only free variables in the residual expressions end up in the resulting environments.) Here, $\curlyvee$ is the merging operator defined as: $\mu \curlyvee \mu' = \mu \cup \mu'$ if there is no x such that $\mu(x) \neq \mu'(x)$. For example, $\{x = 3\} \curlyvee \{y = 4\} = \{x = 3, y = 4\}$, and $\{x = 3, y = 4\} \curlyvee \{y = 4\} = \{x = 3, y = 4\}$, but $\{x = 3, y = 2\} \curlyvee \{y = 4\}$ is undefined.

The most interesting rules are for **<u>case</u>**. In the *get* direction, it is not different from the ordinary **case** except that exit conditions are asserted, as shown in Fig. 6. We use the following predicate for pattern matching.

$$\mathit{match}(p_k, v_0, \mu_k) \;=\; (p_k\mu_k = v_0) \wedge (\mathsf{dom}(\mu_k) = \mathsf{fv}(p_k))$$

Here, we abuse the notation to write $p_k\mu_k$ for the value obtained from p_k by replacing the free variables x in p_k with $\mu_k(x)$. One might notice that we have the disjoint union $\mu \uplus \mu_i$ in Fig. 6 where μ_i holds the values of the variables in p_i, as we assume α-renaming of bound variables that is consistent in *get* and *put*. Recall that p_1 and p_2 are assumed not to overlap, and hence the evaluation is deterministic. Note that the reconciliation functions E_i'' are untouched by the rule.

The *put* evaluation rule of **<u>case</u>** shown in Fig. 6 is more involved. In addition to checking which branch should be chosen by using exit conditions, we need two rules to handle the cases with and without branch switching. Basically,

$$\frac{\mu \vdash_{\mathrm{G}} E_0 \Rightarrow v_0 \quad \mathit{match}(p_i, v_0, \mu_i) \quad e_i \Downarrow_{\mathrm{U}} E_i \quad \mu \uplus \mu_i \vdash_{\mathrm{G}} E_i \Rightarrow v \quad E'_i\ v \Downarrow_{\mathrm{U}} \mathsf{True}}{\mu \vdash_{\mathrm{G}} \underline{\mathbf{case}}\ E_0\ \mathbf{of}\ \left\{p_i \to e_i\ \underline{\mathbf{with}}\ E'_i\ \underline{\mathbf{by}}\ E''_i\right\}_{i=1,2} \Rightarrow v}$$

$$\frac{\begin{array}{c}\mu \vdash_{\mathrm{G}} E_0 \Rightarrow v_0 \quad \mathit{match}(p_i, v_0, \mu_i) \quad E'_i\ v \Downarrow_{\mathrm{U}} \mathsf{True} \quad e_i \Downarrow_{\mathrm{U}} E_i \\ \mu \uplus \mu_i \vdash_{\mathrm{P}} v \Leftarrow E_i \dashv \mu' \uplus_{\mathsf{dom}(\mu),\mathsf{dom}(\mu_i)} \mu'_i \quad v'_0 = p_i(\mu'_i \lhd \mu_i) \quad \mu \vdash_{\mathrm{P}} v'_0 \Leftarrow E_0 \dashv \mu'_0\end{array}}{\mu \vdash_{\mathrm{P}} v \Leftarrow \underline{\mathbf{case}}\ E_0\ \mathbf{of}\ \left\{p_i \to e_i\ \underline{\mathbf{with}}\ E'_i\ \underline{\mathbf{by}}\ E''_i\right\}_{i=1,2} \dashv \mu'_0 \curlyvee \mu'}$$

$$\frac{\begin{array}{c}\mu \vdash_{\mathrm{G}} E_0 \Rightarrow v_0 \quad \mathit{match}(p_i, v_0, \mu_i) \quad E'_i\ v \Downarrow_{\mathrm{U}} \mathsf{False} \quad j = 3 - i \quad E'_j\ v \Downarrow_{\mathrm{U}} \mathsf{True} \quad e_j \Downarrow_{\mathrm{U}} E_j \\ E''_j\ v_0\ v \Downarrow_{\mathrm{U}} u_0 \quad \mathit{match}(p_j, u_0, \mu_j) \\ \mu \uplus \mu_j \vdash_{\mathrm{P}} v \Leftarrow E_j \dashv \mu' \uplus_{\mathsf{dom}(\mu),\mathsf{dom}(\mu_j)} \mu'_j \quad v'_0 = p_j(\mu'_j \lhd \mu_j) \quad \mu \vdash_{\mathrm{P}} v'_0 \Leftarrow E_0 \dashv \mu'_0\end{array}}{\mu \vdash_{\mathrm{P}} v \Leftarrow \underline{\mathbf{case}}\ E_0\ \mathbf{of}\ \left\{p_i \to e_i\ \underline{\mathbf{with}}\ E'_i\ \underline{\mathbf{by}}\ E''_i\right\}_{i=1,2} \dashv \mu'_0 \curlyvee \mu'}$$

Fig. 6. *get*- and *put*-Evaluation of **case**: we write $\mu \uplus_{X,Y} \mu'$ to ensure that $\mathsf{dom}(\mu) \subseteq X$ and $\mathsf{dom}(\mu') \subseteq Y$.

the branch to be taken in the backwards direction is decided first, by the *get*-evaluation of the case condition E_0 and the checking of the exit condition E'_i against the updated view v. After that, the body of the chosen branch e_i is firstly uni-directionally evaluated, and then its residual expression E_i is *put*-evaluated. The last step is *put*-evaluation of the case-condition E_0. When branch switching happens, there is the additional step of applying the reconciliation function E''_j.

Note the use of operator $\lhd$ in computing the updated case condition v'_0.

$$(\mu' \lhd \mu)(x) = \begin{cases} \mu'(x) & \text{if } x \in \mathsf{dom}(\mu') \\ \mu(x) & \text{otherwise} \end{cases}$$

Recall that in the beginning of this subsection, we discussed our approach of avoiding conflicts by producing environments with only relevant variables. This means the μ'_i above contains only variables that appear free in E_i, which may or may not be all the variables in p_i. Since this is the point where these variables are introduced, we need to supplement μ'_i with μ_i from the original pattern matching so that p_i can be properly instantiated.

Construction of Lens. Let us write $\mathcal{L}_0[\![E]\!]$ for a lens between value environments and values, defined as:

$$\begin{array}{ll} \mathit{get}\ \mathcal{L}_0[\![E]\!]\ \mu = v & \text{if } \mu \vdash_{\mathrm{G}} E \Rightarrow v \\ \mathit{put}\ \mathcal{L}_0[\![E]\!]\ \mu\ v = \mu' & \text{if } \mu \vdash_{\mathrm{P}} v \Leftarrow E \dashv \mu' \end{array}$$

Then, we can define the lens $\mathcal{L}[\![e]\!]$ induced from e (a closed function expression), where $e\ x \Downarrow_{\mathrm{U}} E$ for some fresh variable x.

$$\begin{array}{l} \mathit{get}\ \mathcal{L}[\![e]\!]\ s = \mathit{get}\ \mathcal{L}_0[\![E]\!]\ \{x = s\} \\ \mathit{put}\ \mathcal{L}[\![e]\!]\ s\ v = (\mu' \lhd \{x = s\})(x) \quad \text{where } \mu' = \mathit{put}\ \mathcal{L}_0[\![E]\!]\ \{x = s\}\ v \end{array}$$

Actually, `:get` and `:put` in Sect. 2 are realised by *get* $\mathcal{L}[\![e]\!]$ and *put* $\mathcal{L}[\![e]\!]$.

4.3 Correctness

We establish the correctness of HOBiT Core: $\mathcal{L}[\![e]\!] \in \mathit{Lens}\ [\![\sigma]\!]\ [\![\tau]\!]$ is well-behaved for closed e of type $\mathbf{B}\sigma \to \mathbf{B}\tau$. Recall that $\mathit{Lens}\ S\ V$ is a set of lenses ℓ, where $\mathit{get}\ \ell \in S \to V$ and $\mathit{put}\ \ell \in S \to V \to S$. We only provide proof sketches in this subsection due to space limitation.

$\preceq$-well-behavedness. Recall that in the previous subsection, we allow environments to be weakened during *put*-evaluation. Since not all variables in a source may appear in the view, during some intermediate evaluation steps (for example within **case**-branches) the weakened environment may not be sufficient to fully construct a new source. Recall that, in $\mu \vdash_{\mathrm{P}} v \Leftarrow e \dashv \mu'$, $\mathsf{dom}(\mu')$ can be smaller than $\mathsf{dom}(\mu)$, a gap that is fixed at a later stage of evaluation by merging ($\curlyvee$) and defaulting ($\triangleleft$) with other environments. This technique reduces conflicts, but at the same time complicates the compositional reasoning of correctness. Specifically, due to the potentially missing information in the intermediate environments, well-behavedness may be temporally broken during evaluation. Instead, we use a variant of well-behavedness that is weakening aware, which will then be used to establish the standard well-behavedness for the final result.

Definition 1 ($\preceq$-well-behavedness). Let $(S, \preceq)$ and $(V, \preceq)$ be partially-ordered sets. A lens $\ell \in \mathit{Lens}\ S\ V$ is called *$\preceq$-well-behaved* if it satisfies

$$\mathit{get}\ \ell\ s = v \implies v \text{ is maximal} \wedge (\forall v'.\ v' \preceq v \Longrightarrow \mathit{put}\ \ell\ s\ v' \preceq s) \qquad (\preceq\textbf{-Acceptability})$$

$$\mathit{put}\ \ell\ s\ v = s' \implies (\forall s''.\ s' \preceq s'' \Longrightarrow v \preceq \mathit{get}\ \ell\ s'') \qquad (\preceq\textbf{-Consistency})$$

for any $s, s' \in S$ and $v \in V$, where s is maximal. □

We write $\mathit{Lens}^{\preceq\mathrm{wb}}\ S\ V$ for the set of lenses in $\mathit{Lens}\ S\ V$ that are $\preceq$-well-behaved. In this section, we only consider the case where S and V are value environments and first-order values, where value environments are ordered by weakening ($\mu \preceq \mu'$ if $\mu(x) = \mu'(x)$ for all $x \in \mathsf{dom}(\mu)$), and $(\preceq) = (=)$ for first-order values. In Sect. 5.2 we consider a slightly more general situation.

The $\preceq$-well-behavedness is a generalisation of the ordinary well-behavedness, as it coincides with the ordinary well-behavedness when $(\preceq) = (=)$.

Theorem 1. *For S and V with $(\preceq) = (=)$, a lens $\ell \in \mathit{Lens}\ S\ V$ is $\preceq$-well-behaved iff it is well-behaved.* □

Kripke Logical Relation. The key step to prove the correctness of HOBiT Core is to prove that $\mathcal{L}_0[\![E]\!]$ is always $\preceq$-well-behaved if E is an evaluation result of a well-typed expression e. The basic idea is to prove this by logical relation that expression e of type $\mathbf{B}\sigma$ under the context Δ is evaluated to E, assuming termination, such that $\mathcal{L}_0[\![E]\!]$ is a $\preceq$-well-behaved lens between $[\![\Delta]\!]$ and $[\![\sigma]\!]$.

Usually a logical relation is defined only by induction on the type. In our case, as we need to consider Δ in the interpretation of $\mathbf{B}\sigma$, the relation should be indexed by Δ too. However, naive indexing does not work due to substitutions.

For example, we could define a (unary) relation $\mathcal{E}_\Delta(\mathbf{B}\sigma)$ as a set of expressions that evaluate to "good" (i.e., $\preceq$-well-behaved) lenses between (the semantics of) Δ and σ, and $\mathcal{E}_\Delta(\mathbf{B}\sigma \to \mathbf{B}\tau)$ as a set of expressions that evaluate to "good" functions that map good lenses between Δ and σ to those between Δ and τ. This naive relation, however, does not respect substitution, which can substitute a value obtained from an expression typed under Δ to a variable typed under Δ' such that $\Delta \subseteq \Delta'$, where Δ and Δ' need not be the same. With the naive definition, good functions at Δ need not be good functions at Δ', as a good lens between Δ' and σ is not always a good lens between Δ and σ.

To remedy the situation, inspired by the denotation semantics in [24], we use Kripke logical relations [18] where worlds are Δs.

Definition 2. We define the set $\mathcal{E}_\Delta[\![A]\!]$ of expressions, the set $\mathcal{R}_\Delta[\![A]\!]$ of residual expressions, the set $[\![\sigma]\!]$ of values and the set $[\![\Delta]\!]$ of value environments as below.

$$\begin{aligned}
\mathcal{E}_\Delta[\![A]\!] &= \{e \mid \forall E.\ e \Downarrow_{\mathrm{U}} E \text{ implies } E \in \mathcal{R}_\Delta[\![A]\!]\} \\
\mathcal{R}_\Delta[\![Bool]\!] &= \{\mathsf{True}, \mathsf{False}\} \\
\mathcal{R}_\Delta[\![[A]]\!] &= List\ \mathcal{R}_\Delta[\![A]\!] \\
\mathcal{R}_\Delta[\![\mathbf{B}\sigma]\!] &= \{E \mid \forall \Delta'.\ \Delta \subseteq \Delta' \text{ implies } \mathcal{L}_0[\![E]\!] \in Lens^{\preceq\mathrm{wb}}\ [\![\Delta']\!]\ [\![\sigma]\!]\} \\
\mathcal{R}_\Delta[\![A \to B]\!] &= \{F \mid \forall \Delta'.\ \Delta \subseteq \Delta' \text{ implies } (\forall E \in \mathcal{R}_{\Delta'}[\![A]\!].\ F\ E \in \mathcal{E}_{\Delta'}[\![B]\!])\} \\
[\![Bool]\!] &= \{\mathsf{True}, \mathsf{False}\} \\
[\![[\sigma]]\!] &= List\ [\![\sigma]\!] \\
[\![\Delta]\!] &= \{\mu \mid \mathsf{dom}(\mu) \subseteq \mathsf{dom}(\Delta) \text{ and } \forall x \in \mathsf{dom}(\mu). \mu(x) \in [\![\Delta(x)]\!]\}
\end{aligned}$$

Here, for a set S, $List\ S$ is inductively defined as: $[\,] \in List\ S$, and $s : t \in List\ S$ for all $s \in S$ and $t \in List\ S$. □

The notable difference from ordinary logical relations is the definition of $\mathcal{R}_\Delta[\![A \to B]\!]$ where we consider an arbitrary Δ' such that $\Delta \subseteq \Delta'$. This is the key to state $\mathcal{R}_\Delta[\![A]\!] \subseteq \mathcal{R}_{\Delta'}[\![A]\!]$ if $\Delta \subseteq \Delta'$. Notice that $[\![\sigma]\!] = \mathcal{R}_\Delta[\![\sigma]\!]$ for any Δ.

We have the following lemmas.

Lemma 1. *If* $\Delta \subseteq \Delta'$, $v \in \mathcal{R}_\Delta[\![A]\!]$ *implies* $v \in \mathcal{R}_{\Delta'}[\![A]\!]$. □

Lemma 2. $x \in \mathcal{R}_\Delta[\![\mathbf{B}\sigma]\!]$ *for any* Δ *such that* $\Delta(x) = \sigma$. □

Lemma 3. *For any* σ *and* Δ, $\underline{\mathsf{True}}, \underline{\mathsf{False}} \in \mathcal{R}_\Delta[\![\mathbf{B}Bool]\!]$ *and* $\underline{[\,]} \in \mathcal{R}_\Delta[\![\mathbf{B}[\sigma]]\!]$. □

Lemma 4. *If* $E_1 \in \mathcal{R}_\Delta[\![\mathbf{B}\sigma]\!]$ *and* $E_2 \in \mathcal{R}_\Delta[\![\mathbf{B}[\sigma]]\!]$, *then* $E_1 \underline{:} E_2 \in \mathcal{R}_\Delta[\![\mathbf{B}[\sigma]]\!]$. □

Lemma 5. *Let* σ *and* τ *be pure types and* Δ *a pure type environment. Suppose that* $e_i \in \mathcal{E}_{\Delta \uplus \Delta_i}[\![\tau]\!]$ *for* $\Delta_i \vdash p_i : \sigma$ $(i = 1, 2)$, *and that* $E_0 \in \mathcal{R}_\Delta[\![\mathbf{B}\sigma]\!]$, $E'_1, E'_2 \in \mathcal{R}_\Delta[\![\tau \to Bool]\!]$ *and* $E''_1, E''_2 \in \mathcal{R}_\Delta[\![\sigma \to \tau \to \sigma]\!]$. *Then,* $\underline{\mathbf{case}}$ E_0 $\underline{\mathbf{of}}$ $\{p_i \to e_i\ \underline{\mathbf{with}}\ E'_i\ \underline{\mathbf{by}}\ E''_i\}_{i=1,2} \in \mathcal{R}_\Delta[\![\mathbf{B}\tau]\!]$.

Proof (Sketch). The proof itself is straightforward by case analysis. The key property is that *get* and *put* use the same branches in both proofs of $\preceq$-**Acceptability** and $\preceq$-**Consistency**. Slight care is required for unidirectional evaluations of e_1 and e_2, and applications of E_1', E_2', E_1'' and E_2''. However, the semantics is carefully designed so that in the proof of $\preceq$-**Acceptability**, unidirectional evaluations that happen in *put* have already happened in the evaluation of *get*, and a similar discussion applies to $\preceq$-**Consistency**. $\square$

As a remark, recall that we assumed α-renaming of p_i so that the disjoint unions ($\uplus$) in Fig. 6 succeed. This renaming depends on the μs received in *get* and *put* evaluations, and can be realised by using de Bruijn levels.

Lemma 6 (Fundamental Lemma). *For $\Gamma; \Delta \vdash e : A$, for any Δ' with $\Delta \subseteq \Delta'$ and $E_x \in \mathcal{R}_{\Delta'}[\![\Gamma(x)]\!]$, we have $e[E_x/x]_x \in \mathcal{E}_{\Delta'}[\![A]\!]$.*

Proof (Sketch). We prove the lemma by induction on typing derivation. For bidirectional constructs, we just apply the above lemmas appropriately. The other parts are rather routine. $\square$

Now we are ready to state the correctness of our construction of lenses.

Corollary 1. *If $\varepsilon; \varepsilon \vdash e : \mathbf{B}\sigma \to \mathbf{B}\tau$, then $e\ x \in \mathcal{E}_{\{x:\sigma\}}[\![\mathbf{B}\tau]\!]$.* $\square$

Lemma 7. *If $e \in \mathcal{E}_{\{x:\sigma\}}[\![\mathbf{B}\tau]\!]$, $\mathcal{L}[\![e]\!]$ (if defined) is in $Lens^{\preceq \mathrm{wb}}\ [\![\sigma]\!]\ [\![\tau]\!]$ (and thus well-behaved by Theorem 1).* $\square$

Theorem 2. *If $\varepsilon; \varepsilon \vdash e : \mathbf{B}\sigma \to \mathbf{B}\tau$, then $\mathcal{L}[\![e]\!] \in Lens\ [\![\sigma]\!]\ [\![\tau]\!]$ (if defined) is well-behaved.* $\square$

5 Extensions

Before presenting a larger example, we discuss a few extensions of HOBiT Core which facilitate programming.

5.1 In-Language Lens Definition

In HOBiT programming, it is still sometimes useful to allow manually defined primitive lenses (i.e., lenses constructed from independently specified *get*/*put* functions), for backwards compatibility and also for programs with relatively simple computation logic but complicated backwards behaviours. This feature is supported by the construct $\underline{\textbf{appLens}}\ e_1\ e_2\ e_3$ in HOBiT. For example, we can write $incB\ x = \underline{\textbf{appLens}}\ (\lambda s.s+1)\ (\lambda\_.\lambda v.v-1)\ x$ to define a bidirectional increment function $incB :: \mathbf{B}Int \to \mathbf{B}Int$. Note that for simplicity we require the additional expression x (represented by e_3 in the general case) to convert between normal functions and lenses. The typing rule for $\underline{\mathbf{appLens}}\ e_1\ e_2\ e_3$ is as below.

$$\frac{\Gamma;\Delta \vdash e_1 : \sigma \to \tau \quad \Gamma;\Delta \vdash e_2 : \sigma \to \tau \to \sigma \quad \Gamma;\Delta \vdash e_3 : \mathsf{B}\sigma}{\Gamma;\Delta \vdash \underline{\mathbf{appLens}}\ e_1\ e_2\ e_3 : \mathsf{B}\tau}$$

Accordingly, we add the following unidirectional evaluation rule.

$$\frac{e_i \Downarrow_{\mathrm{U}} E_i \quad (i = 1, 2, 3)}{\underline{\mathbf{appLens}}\ e_1\ e_2\ e_3 \Downarrow_{\mathrm{U}} \underline{\mathbf{appLens}}\ E_1\ E_2\ E_3}$$

Also, we add the following *get*/*put* evaluation rules for $\underline{\mathbf{appLens}}$.

$$\frac{\mu \vdash_{\mathrm{G}} E_3 \Rightarrow v \quad E_1\ v \Downarrow_{\mathrm{U}} u}{\mu \vdash_{\mathrm{G}} \underline{\mathbf{appLens}}\ E_1\ E_2\ E_3 \Rightarrow u} \qquad \frac{\mu \vdash_{\mathrm{G}} E_3 \Rightarrow v \quad E_2\ v\ u' \Downarrow_{\mathrm{U}} v' \quad \mu \vdash_{\mathrm{P}} v' \Leftarrow E_3 \dashv \mu'}{\mu \vdash_{\mathrm{P}} u' \Leftarrow \underline{\mathbf{appLens}}\ E_1\ E_2\ E_3 \dashv \mu'}$$

Notice that $\underline{\mathbf{appLens}}\ e_1\ e_2\ e_3$ is "good" if e_3 is so, i.e., $\underline{\mathbf{appLens}}\ e_1\ e_2\ e_3 \in \mathcal{E}_\Delta[\![\mathsf{B}\tau]\!]$ if $e_3 \in \mathcal{E}_\Delta[\![\mathsf{B}\sigma]\!]$, provided that the *get*/*put* pair (e_1, e_2) is well-behaved.

5.2 Lens Combinators as Language Constructs

In this paper, we have focused on the $\underline{\mathbf{case}}$ construct, which is inspired by the *cond* combinator [7]. Although *cond* is certainly an important lens combinator, it is not the only one worth considering. Actually, we can obtain language constructs from a number of lens combinators including those that take care of alignment [2]. For the sake of demonstration, we outline the derivation of a simpler example $comb \in Lens\ [\![\sigma]\!]\ [\![\tau]\!] \to Lens\ [\![\sigma']\!]\ [\![\tau']\!]$. As the construction depends solely on types, we purposely leave the combinator abstract.

A naive way of lifting combinators can already be found in [21,23]. For example, for *comb*, we might prepare the construct $\underline{\mathbf{comb}}_{\mathrm{bad}}$ with the following typing rule (where ε is the empty environment):

$$\frac{\varepsilon;\varepsilon \vdash e : \mathsf{B}\sigma \to \mathsf{B}\tau \quad \Gamma;\Delta \vdash e' : \mathsf{B}\tau'}{\Gamma;\Delta \vdash \underline{\mathbf{comb}}_{\mathrm{bad}}\ e\ e' : \mathsf{B}\tau'}$$

Notice that in this version e is required to be closed so that we can turn the function directly into a lens by $\mathcal{L}[\![-]\!]$, and the evaluation of $\underline{\mathbf{comb}}_{\mathrm{bad}}$ can then be based on standard lens composition: $\mathcal{L}_0[\![\underline{\mathbf{comb}}_{\mathrm{bad}}\ E\ E']\!] = comb\ \mathcal{L}[\![E]\!] \mathbin{\hat{\circ}} \mathcal{L}_0[\![E']\!]$ (we omit the straightforward concrete evaluation rules), where E and E' is the unidirectional evaluation results of e and e' (notice that a residual expression is also an expression), and $\hat{\circ}$ is the lens composition combinator [7] defined by:

$$\begin{array}{l} (\hat{\circ}) \in Lens\ B\ C \to Lens\ A\ B \to Lens\ A\ C \\ get\ (\ell_2 \mathbin{\hat{\circ}} \ell_1)\ a \quad\ = get\ \ell_2\ (get\ \ell_1\ a) \\ put\ (\ell_2 \mathbin{\hat{\circ}} \ell_1)\ a\ c' = put\ \ell_1\ a\ (put\ \ell_2\ (get\ \ell_1\ a)\ c') \end{array}$$

The combinator preserves $\preceq$-well-behavedness, and thus $\underline{\mathbf{comb}}_{\mathrm{bad}}$ guarantees correctness. However, as discussed extensively in the case of $\underline{\mathbf{case}}$, this "closedness" requirements prevents flexible use of variables and creates a major obstacle in programming.

So instead of the plain *comb*, we shall assume a parameterised version $\mathit{pcomb} \in \mathit{Lens}\ (T \times [\![\sigma]\!])\ [\![\tau]\!] \to \mathit{Lens}\ (T \times [\![\sigma']\!])\ [\![\tau']\!]$ that allows each source to have an extra component T, which is expected to be kept track of by the combinator without modification. Here T is assumed to have a partial merging operator $(\curlyvee) \in T \to T \to T$ and a minimum element, and *pcomb* may use these facts in its definition. By using *pcomb*, we can give a corresponding language construct **comb** with a binder, typed as follows.

$$\frac{\Gamma; \Delta, x : \sigma \vdash e : \mathbf{B}\tau \quad \Gamma; \Delta \vdash e' : \mathbf{B}\sigma'}{\Gamma; \Delta \vdash \underline{\mathbf{comb}}\ (x.e)\ e' : \mathbf{B}\tau'}$$

We give its unidirectional evaluation rule as

$$\frac{e \Downarrow_{\mathrm{U}} E \quad e' \Downarrow_{\mathrm{U}} E'}{\underline{\mathbf{comb}}\ (x.e)\ e' \Downarrow_{\mathrm{U}} \underline{\mathbf{comb}}\ E\ E'}$$

We omit the *get*/*put* evaluation rules, which are straightforwardly obtained from the following equation.

$$\mathcal{L}_0[\![\underline{\mathbf{comb}}\ E\ E']\!] = \mathit{pcomb}\ (\mathit{unEnv}_x\ \mathcal{L}_0[\![E]\!])\ \hat{\circ}\ \langle \mathit{idL}, \mathcal{L}_0[\![E']\!]\rangle$$

where $\mathit{unEnv}_x \in \mathit{Lens}\ ([\![\Delta \uplus \{x : \sigma\}]\!])\ [\![\tau]\!] \to \mathit{Lens}\ ([\![\Delta]\!] \times [\![\sigma]\!])\ [\![\tau]\!]$ and $\langle -, - \rangle \in \mathit{Lens}\ [\![\Delta]\!]\ A \to \mathit{Lens}\ [\![\Delta]\!]\ B \to \mathit{Lens}\ [\![\Delta]\!]\ (A \times B)$ are lens combinators defined for any Δ as:

$$\begin{array}{l}
\mathit{get}\ (\mathit{unEnv}_x\ \ell)\ (\mu, v) \quad = \mathit{get}\ \ell\ (\mu \uplus \{x = v\}) \\
\mathit{put}\ (\mathit{unEnv}_x\ \ell)\ (\mu, v)\ u = (\mu', v') \\
\qquad \mathbf{where}\ \mu' \uplus \{x = v'\} = (\mathit{put}\ \ell\ (\mu \uplus \{x = v\})\ v) \triangleleft \{x = v\} \\[1ex]
\mathit{get}\ \langle \ell_1, \ell_2 \rangle\ \mu \qquad\quad = (\mathit{get}\ \ell_1\ \mu, \mathit{get}\ \ell_2\ \mu) \\
\mathit{put}\ \langle \ell_1, \ell_2 \rangle\ \mu\ (a, b) = \mathit{put}\ \ell_1\ \mu\ a \curlyvee \mathit{put}\ \ell_2\ \mu\ b
\end{array}$$

Both combinators preserve $\preceq$-well-behavedness, where we assume the component-wise ordering on pairs. No "closedness" requirement is imposed on e in this version. From the construct, we can construct a higher-order function $\lambda f.\lambda z.\underline{\mathbf{comb}}\ (x.f\ x)\ z : (\mathbf{B}\sigma \to \mathbf{B}\tau) \to \mathbf{B}\sigma' \to \mathbf{B}\tau'$. That is, in HOBiT, lens combinators are just higher-order functions, as long as they permit the above-mentioned parameterisation. This observation means that we are able to systematically derive language constructs from lens combinators; as a matter of fact, the semantics of **case** is derived from a variant of the *cond* combinator [7].

Even better, the parametrised *pcomb* can be systematically constructed from the definition of *comb*. For *comb*, it is typical that $\mathit{get}\ (\mathit{comb}\ \ell)$ only uses $\mathit{get}\ \ell$, and $\mathit{put}\ (\mathit{comb}\ \ell)$ uses $\mathit{put}\ \ell$; that is, *comb* essentially consists of two functions of types $([\![\sigma]\!] \to [\![\tau]\!]) \to ([\![\sigma']\!] \to [\![\tau']\!])$ and $([\![\sigma]\!] \to [\![\tau]\!] \to [\![\sigma]\!]) \to ([\![\sigma']\!] \to [\![\tau']\!] \to [\![\sigma']\!])$. Then, we can obtain *pcomb* of the above type merely by "monad"ifying the two functions: using the reader monad $T \to -$ for the former and the composition of the reader and writer monads $T \to (-, T)$ backwards for the latter suffice to construct *pcomb*.

A remaining issue is to ensure that *pcomb* preserves $\preceq$-well-behavedness, which ensures $\underline{\mathbf{comb}}\,(x.e)\;e' \in \mathcal{E}_\Delta[\![\mathsf{B}\tau']\!]$ under the assumptions $e \in \mathcal{E}_{\Delta \uplus \{x:\sigma\}}[\![\mathsf{B}\tau]\!]$ and $e' \in \mathcal{E}_\Delta[\![\mathsf{B}\sigma']\!]$. Currently, such a proof has to be done manually, even though *comb* preserves well-behavedness and *pcomb* is systematically constructed. Whether we can lift the correctness proof for *comb* to *pcomb* in a systematic way will be an interesting future exploration.

5.3 Guards

Guards used for branching are merely syntactic sugar in ordinary unidirectional languages such as Haskell. But interestingly, they actually increase the expressive power of HOBiT, by enabling inspection of updatable values without making the inspection functions bidirectional.

For example, Glück and Kawabe's reversible equivalence check [10] can be implemented in HOBiT as follows.

$$\begin{array}{l}
\mathit{eqCheck} :: \mathsf{B}\sigma \to \mathsf{B}\sigma \to \mathsf{B}(\mathit{Either}\;(\sigma,\sigma)\;\sigma) \\
\mathit{eqCheck}\;x\;y = \underline{\mathbf{case}}\;\underline{(}x,y\underline{)}\;\underline{\mathbf{of}} \\
\quad (x',y') \mid x' == y' \quad \to \underline{\mathsf{Right}}\;x' \qquad \underline{\mathbf{with}}\;\mathit{isRight}\;\underline{\mathbf{by}}\;(\lambda\_.\lambda(\mathsf{Right}\;x).(x,x)) \\
\quad (x',y') \mid \mathit{otherwise} \to \underline{\mathsf{Left}}\;\underline{(}x',y'\underline{)}\;\underline{\mathbf{with}}\;\mathit{isLeft}\;\;\underline{\mathbf{by}}\;(\lambda\_.\lambda(\mathsf{Left}\;(x,y)).(x,y))
\end{array}$$

Here, $\underline{(}-,-\underline{)}$ is the bidirectional version of the pair constructor. The exit condition *isRight* checks whether a value is headed by the constructor Right, and *isLeft* by Left. Notice that the backwards transformation of *eqCheck* fails when the updated view is Left (v,v) for some v.

5.4 Syntax Sugar for Reconciliation Functions

In the general form, reconciliation functions take in two arguments for the computation of the new source. But as we have seen, very often the arguments are not used in the definition and therefore redundant. This observation motivates the following syntax sugar.

$$p \to e\;\underline{\mathbf{with}}\;e'\;\underline{\mathbf{default}}\;\{x_1 = e''_1; \ldots; x_n = e''_n\}$$

Here, $x_1, \ldots, x_n$ are the free variables in p. This syntax sugar is translated as:

$$p \to e\;\underline{\mathbf{with}}\;e'\;\underline{\mathbf{by}}\;\lambda\_.\lambda\_.p[e''_1/x_1, \ldots, e''_n/x_n]$$

Furthermore, it is also possible to automatically derive some default values from their types. This idea can be effectively implemented if we extend HOBiT with type classes.

5.5 Inference of Exit Conditions

It is possible to infer exit conditions from their surrounding contexts; an idea that has been studied in the literature of invertible programming [11,20], and may benefit from range analysis.

Our prototype implementation adopts a very simple inference that constructs an exit condition $\lambda x.\mathbf{case}\ x\ \mathbf{of}\ \{p_e \to \mathsf{True}; \_ \to \mathsf{False}\}$ for each branch, where p_e is the skeleton of the branch body e, constructed by replacing bidirectional constructors with the unidirectional counterparts, and non-constructor expressions with $\_$. For example, from $a \mathbin{\underline{:}} \mathit{appendB}\ x'\ y$, we obtain the pattern $\_ : \_$. This embarrassingly simple inference has proven to be handy for developing larger HOBiT programs as we will see in Sect. 6.

6 An Involved Example: Desugaring

In this section, we demonstrate the programmability of HOBiT using the example of bidirectional desugaring [26]. Desugaring is a standard process for most programming languages, and making it bidirectional allows information in desugared form to be propagated back to the surface programs. It is argued convincingly in [26] that such bidirectional propagation (coined *resugaring*) is effective in mapping reduction sequences of desugared programs into those of the surface programs.

Let us consider a small programming language that consists of **let**, **if**, Boolean constants, and predefined operators.

$$\begin{array}{l}\mathbf{data}\ E = \mathsf{ELet}\ E\ E \mid \mathsf{EVar}\ \mathit{Int} \mid \mathsf{EIf}\ E\ E\ E \mid \mathsf{ETrue} \mid \mathsf{EFalse} \mid \mathsf{EOp}\ \mathit{Name}\ [E] \\ \mathbf{type}\ \mathit{Name} = \mathit{String}\end{array}$$

Variables are represented as de Bruijn indices.

Some operators in this language are syntactic sugar. For example, we may want to desugar

$$\mathsf{EOp}\ \texttt{"not"}\ [e] \qquad \text{as} \qquad \mathsf{EIf}\ e\ \mathsf{EFalse}\ \mathsf{ETrue}.$$

Also, e_1 `||` e_2 can be transformed to **let** $x = e_1$ **in if** x **then** x **else** e_2, which in our mini-language is the following.

$$\mathsf{EOp}\ \texttt{"or"}\ [e_1, e_2] \qquad \text{as} \qquad \mathsf{ELet}\ e_1\ (\mathsf{EIf}\ (\mathsf{EVar}\ 0)\ (\mathsf{EVar}\ 0)\ (\mathit{shift}\ 0\ e_2))$$

Here, *shift* n is the standard shifting operator for de Brujin indexed-term that increments the variables that have indices greater than n (these variables are "free" in the given expression). We will program a bidirectional version of the above desugaring process in Figs. 7 and 8, with the particular goal of keeping the result of a backward execution as close as possible to the original sugared form (so that it is not merely a "decompilation" in the sense that the original source has to be consulted).

```
composB :: (BE → BE) → BE → BE
composB f x = case x of
  EIf e1 e2 e3 → EIf (f e1) (f e2) (f e3)       by recE
  ELet e1 e2   → ELet (f e1) (f e2)             by recE
  EVar n       → EVar n                         by recE
  ETrue        → ETrue
  EFalse       → EFalse
  EOp n es     → EOp n (mapB ETrue f es) by recE
mapB :: a → (Ba → Bb) → B[a] → B[b]
mapB def z = case z of
  []    → []
  a : x → f a : mapB def x default {a = def; x = []}
```

```
recE :: E → E → E
recE e (EIf _ _ _) = EIf ETrue e e
recE e (ELet _ _)  = ELet e (shift 0 e)
recE e (EOp n _)   = toOp n e
recE e e'          = e'
toOp :: Name → E → E
toOp n e =
  let k = fromJust (lookup n arities)
  in EOp (replicate k e)
```

Fig. 7. *composB*: a useful building block

```
shiftB :: Int → BE → BE
shiftB n e = case e of
  ELet e1 e2        → ELet (shiftB n e1) (shiftB (n + 1) e2)   default {e1 = ETrue; e2 = EFalse}
  EVar m | m < n → EVar m                    with varLT n       default m = 0
  EVar m | m ≥ n → EVar (incB m)             with varGT n       default m = n + 1
  e'                → composB (shiftB n) e' with nonLetVar by recE
desugarB :: BE → BE
desugarB e = case e of
  EOp "or" [e1, e2] → ELet (desugarB e1) (EIf (EVar 0) (EVar 0) (desugarB (shiftB 0 e2)))
                                          by (λs.λ_.toOp "or" s)
  EOp "not" [e]     → EIf e EFalse ETrue  by (λs.λ_.toOp "not" s)
  e'                → composB desugarB e' by recE

varLT n (EVar m) = m < n           nonLetVar (ELet _ _) = False
varLT n _        = False           nonLetVar (EVar _)   = False
varGT n (EVar m) = m > n           nonLetVar e          = True
varGT n _        = False
```

Fig. 8. *desugarB*: bidirectional desugring

We start with an auxiliary function *compos* [4] in Fig. 7, which is a useful building block for defining shifting and desugaring. We have omitted the straightforward exit conditions; they will be inferred as explained in Sect. 5.5. The function *mapB* is the bidirectional map. The reconciliation function *recE* tries to preserves as much source structure as possible by reusing the original source *e*. Here, *arities* :: [(*Name*, *Int*)] maps operator names to their arities (i.e. *arities* = [("or", 2), ("not", 1)]). The function *shift* is the standard uni-directional shifting function. We omit its definition as it is similar to the bidirectional version in Fig. 8. Note that **default** is syntactic sugar for reconciliation function introduced in Sect. 5.4. Here, *incB* is the bidirectional increment function defined in Sect. 5.1. Thanks to *composB*, we only need to define the interesting parts in the definitions of *shiftB* and *desugarB*. The reconciliation functions *recE* and *toOp* try to keep as much source information as possible, which enables the behaviour that the backwards execution produces "`not`" and "`or`" in the sugared form only if the original expression has the sugar.

Consider a sugared expression EOp "or" [EOp "not" [ETrue], EOp "not" [EFalse]] as a source *source*.

```
HOBiT> :get desugarB source
ELet (EIf ETrue EFalse ETrue) (EIf (EVar 0) (EVar 0) (EIf EFalse EFalse ETrue)
{- let x = (if True then False else True)
   in if x then x else (if False then False else True) -}
```

The following updated views may be obtained by reductions from the view.

```
{- view₁ ≡ let x = False in if x then x else (if False then False else True) -}
view₁ = ELet EFalse (EIf (EVar 0) (EVar 0) (EIf EFalse EFalse ETrue)
{- view₂ ≡ if False then False else (if False then False else True) -}
view₂ = EIf EFalse EFalse (EIf EFalse EFalse ETrue)
{- view₃ ≡ if False then False else True -}
view₃ = EIf EFalse EFalse ETrue
```

The following are the corresponding backward transformation results.

```
HOBiT> :put desugarB source view₁
EOp "or" [EFalse, EOp "not" [EFalse]]
HOBiT> :put desugarB source view₂
EIf EFalse EFalse (EOp "not" [EFalse]
HOBiT> :put desugarB source view₃
EOp "not" [False]
```

As the AST structure of the view is changed, all of the three cases require branch-switching in the backwards executions; our program handles it with ease. For $view_2$, the top-level expression EIf EFalse EFalse ... does not have a corresponding sugared form. Our program keeps the top level unchanged, and proceeds to the subexpression with correct resugaring, a behaviour enabled by the appropriate use of reconciliation function (the first line of *recE* for this particular case) in *composB*.

If we were to present the above results as the evaluation steps in the surface language, one may argue that the second result above does not correspond to a valid evaluation step in the surface language. In [26], AST nodes introduced in desugaring are marked with the information of the original sugared syntax, and resugaring results containing the marked nodes will be skipped, as they do not correspond to any reduction step in the surface language. The marking also makes the backwards behaviour more predictable and stable for drastic changes on the view, as the desugaring becomes injective with this change. This technique is orthogonal to our exploration here, and may be combined with our approach.

7 Related Work

Controlling Backwards Behaviour. In addition to $put \in S \to V \to S$, many lens languages [3] supply a $create \in V \to S$ (which is in essence a right-inverse of *get*) to be used when the original source data is unavailable. This happens when new data is inserted in the view, which does not have any corresponding source for *put* to execute, or when branch-switching happens but with no reconciliation function available. Being a right-inverse, *create* does not fail (assuming it terminates), but since it is not guided by the original source, the results are more arbitrary. We do not include *create* in HOBiT, as it complicates the system without offering obvious benefits. Our branch-switching facilities are perfectly capable of handling missing source data via reconciliation functions.

Using exit conditions in branching constructs for backwards evaluation can be found in a number of related fields: bidirectional transformation [7], reversible computation [34] and program inversion [11,20]. Our design of **case** is inspired by the *cond* combinator in the lens framework [7] and the `if`-statement in Janus [34]. A similar combinator is *Case* in BiGUL [16], where a branch has a function performing a similar role as an exit condition, but taking the original source in addition. This difference makes *Case* more expressive than *cond*; for example, *Case* can implement matching lenses [2]. Our design of **case** follows *cond* for its relative simplicity, but the same underlying technique can be applied to *Case* as mentioned in Sect. 5.2. In the context of *bidirectionalization* [19,29,30] there is the idea of "Plug-ins" [31] that are similar to reconciliation functions in the sense that source values can be adapted to direct backwards execution.

Applicative Lenses. The applicative lens framework [21,23] provides a way to use λ-abstraction and function application as in normal functional programming to compose lenses. Note that this use of "applicative" refers to the classical applicative (functional) programming style, and is not directly related to `Applicative` functor in Haskell. In this sense, it shares a similar goal to us. But crucially, applicative lens lacks HOBiT's ability to allow λ-bound variables to be used freely, and as a result suffers from the same limitation of lens languages. There are also a couple of technical differences between applicative lens and our work: applicative lens is based on Yoneda embedding while ours is based on separating Γ and Δ and having three semantics (Sect. 4); and applicative lens is implemented as an embedded DSL, while HOBiT is given as a standalone language. Embedded implementation of HOBiT is possible, but a type-correct embedding would expose the handling of environment Δ to programmers, which is undesirable.

Lenses and Their Extensions. As mentioned in Sect. 1, the most common way to construct lenses is by using combinators [3,7,8], in which lenses are treated as opaque objects and composed by using lens combinators. Our goal in this paper is to enhance the programmability of lens programming, while keeping its expressive power as possible. In HOBiT, primitive lenses can be represented as functions on $\mathbf{B}$-typed values (Sect. 5.1), and lens combinators satisfying certain conditions can be represented as language construct with binders (Sect. 5.2), which is at least enough to express the original lenses in [7].

Among extensions of the lens language [2,3,7–9,16,17,27,32], there exists a few that extend the classical lens model [7], namely quotient lenses [8], symmetric lenses [14], and edit-based lenses [15]. A natural question to ask is whether our development, which is based on the classical lenses, can be extended to them. The answer depends on treatment of value environments μ in *get* and *put*. In our semantics, we assume a non-linear system as we can use the same variable in μ any number of times. This requires us to extend the classical lens to allow merging ($\curlyvee$) and defaulting ($\triangleleft$) operations in *put* with $\preceq$-well-behavedness, but makes the syntax and type system of HOBiT simple, and HOBiT free from the design issues of linear programming languages [25]. Such extension of lenses would be applicable to some kinds of lens models, including quotient lenses and symmetric lenses, but its applicability is not clear in general. Also, we want to mention that allowing duplications in bidirectional transformation is still open, as it essentially entails multiple views and the synchronization among them.

8 Conclusion

We have designed HOBiT, a higher-order bidirectional programming language in which lenses are represented as functions and lens combinators are represented as language constructs with binders. The main advantage of HOBiT is that users can program in a style similar to conventional functional programming, while still enjoying the benefits of lenses (i.e., the expressive power and well-behavedness guarantee). This has allowed us to program realistic examples with relative ease.

HOBiT for the first time introduces a truly "functional" way of constructing bidirectional programs, which opens up a new area of future explorations. Particularly, we have just started to look at programming techniques in HOBiT. Moreover, given the resemblance of HOBiT code to that in conventional languages, the application of existing programming tools becomes plausible.

Acknowledgements. We thank Shin-ya Katsumata, Makoto Hamana and Kazuyuki Asada for their helpful comments on the category theory and denotational semantics, from which our formal discussions originate. The work was partially supported by JSPS KAKENHI Grant Numbers 24700020, 15K15966, and 15H02681.

References

1. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Trans. Database Syst. **6**(4), 557–575 (1981). https://doi.org/10.1145/319628.319634
2. Barbosa, D.M.J., Cretin, J., Foster, N., Greenberg, M., Pierce, B.C.: Matching lenses: alignment and view update. In: Hudak, P., Weirich, S. (eds.) ICFP, pp. 193–204. ACM (2010). https://doi.org/10.1145/1863543.1863572
3. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: Necula, G.C., Wadler, P. (eds.) POPL, pp. 407–419. ACM (2008). https://doi.org/10.1145/1328438.1328487
4. Bringert, B., Ranta, A.: A pattern for almost compositional functions. J. Funct. Program. **18**(5–6), 567–598 (2008). https://doi.org/10.1017/S0956796808006898

5. Davies, R., Pfenning, F.: A modal analysis of staged computation. J. ACM **48**(3), 555–604 (2001). https://doi.org/10.1145/382780.382785
6. Fegaras, L.: Propagating updates through XML views using lineage tracing. In: Li, F., Moro, M.M., Ghandeharizadeh, S., Haritsa, J.R., Weikum, G., Carey, M.J., Casati, F., Chang, E.Y., Manolescu, I., Mehrotra, S., Dayal, U., Tsotras, V.J. (eds.) ICDE, pp. 309–320. IEEE (2010). https://doi.org/10.1109/ICDE.2010.5447896
7. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3) (2007). https://doi.org/10.1145/1232420.1232424
8. Foster, J.N., Pilkiewicz, A., Pierce, B.C.: Quotient lenses. In: Hook, J., Thiemann, P. (eds.) ICFP, pp. 383–396. ACM (2008). https://doi.org/10.1145/1411204.1411257
9. Foster, N., Matsuda, K., Voigtländer, J.: Three complementary approaches to bidirectional programming. In: Gibbons, J. (ed.) Generic and Indexed Programming. LNCS, vol. 7470, pp. 1–46. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32202-0_1
10. Glück, R., Kawabe, M.: A program inverter for a functional language with equality and constructors. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 246–264. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40018-9_17
11. Glück, R., Kawabe, M.: Revisiting an automatic program inverter for lisp. SIGPLAN Not. **40**(5), 8–17 (2005). https://doi.org/10.1145/1071221.1071222
12. Hegner, S.J.: Foundations of canonical update support for closed database views. In: Abiteboul, S., Kanellakis, P.C. (eds.) ICDT 1990. LNCS, vol. 470, pp. 422–436. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-53507-1_93
13. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K.: Bidirectionalizing graph transformations. In: Hudak, P., Weirich, S. (eds.) ICFP, pp. 205–216. ACM (2010). https://doi.org/10.1145/1863543.1863573
14. Hofmann, M., Pierce, B.C., Wagner, D.: Symmetric lenses. In: Ball, T., Sagiv, M. (eds.) POPL, pp. 371–384. ACM (2011). https://doi.org/10.1145/1926385.1926428
15. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: Field, J., Hicks, M. (eds.) POPL, pp. 495–508. ACM (2012). https://doi.org/10.1145/2103656.2103715
16. Hu, Z., Ko, H.S.: Principles and practice of bidirectional programming in BiGUL. Oxford Summer School on Bidirectional Transformations (2017). https://bitbucket.org/prl_tokyo/bigul/raw/master/SSBX16/tutorial.pdf. Accessed 18 Oct 2017
17. Hu, Z., Mu, S.-C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. In: Heintze, N., Sestoft, P. (eds.) PEPM, pp. 178–189. ACM (2004). https://doi.org/10.1145/1014007.1014025
18. Jung, A., Tiuryn, J.: A new characterization of lambda definability. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 245–257. Springer, Heidelberg (1993). https://doi.org/10.1007/BFb0037110
19. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: Hinze, R., Ramsey, N. (eds.) ICFP, pp. 47–58. ACM (2007). https://doi.org/10.1145/1291151.1291162
20. Matsuda, K., Mu, S.-C., Hu, Z., Takeichi, M.: A grammar-based approach to invertible programs. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 448–467. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_24
21. Matsuda, K., Wang, M.: Applicative bidirectional programming: mixing lenses and semantic bidirectionalization. J. Funct. Program. Accepted 14 Feb 2018

22. Matsuda, K., Wang, M.: "Bidirectionalization for free" for monomorphic transformations. Sci. Comput. Program. **111**(1), 79–109 (2014). https://doi.org/10.1016/j.scico.2014.07.008
23. Matsuda, K., Wang, M.: Applicative bidirectional programming with lenses. In: Fisher, K., Reppy, J.H. (eds.) ICFP, pp. 62–74. ACM (2015). https://doi.org/10.1145/2784731.2784750
24. Moggi, E.: Functor categories and two-level languages. In: Nivat, M. (ed.) FoSSaCS 1998. LNCS, vol. 1378, pp. 211–225. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053552
25. Morris, J.G.: The best of both worlds: linear functional programming without compromise. In: Garrigue, J., Keller, G., Sumii, E. (eds.) ICFP, pp. 448–461. ACM (2016). https://doi.org/10.1145/2951913.2951925
26. Pombrio, J., Krishnamurthi, S.: Resugaring: lifting evaluation sequences through syntactic sugar. In: O'Boyle, M.F.P., Pingali, K. (eds.) PLDI, pp. 361–371. ACM (2014). https://doi.org/10.1145/2594291.2594319
27. Rajkumar, R., Foster, N., Lindley, S., Cheney, J.: Lenses for web data. ECEASST **57** (2013). https://doi.org/10.14279/tuj.eceasst.57.879
28. Stevens, P.: A landscape of bidirectional model transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 408–424. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88643-3_10
29. Voigtländer, J.: Bidirectionalization for free! (pearl). In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 165–176. ACM (2009). https://doi.org/10.1145/1480881.1480904
30. Voigtländer, J., Hu, Z., Matsuda, K., Wang, M.: Combining syntactic and semantic bidirectionalization. In: Hudak, P., Weirich, S. (eds.) ICFP, pp. 181–192. ACM (2010). https://doi.org/10.1145/1863543.1863571
31. Voigtländer, J., Hu, Z., Matsuda, K., Wang, M.: Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins. J. Funct. Program. **23**(5), 515–551 (2013). https://doi.org/10.1017/S0956796813000130
32. Wang, M., Gibbons, J., Matsuda, K., Hu, Z.: Refactoring pattern matching. Sci. Comput. Program. **78**(11), 2216–2242 (2013). https://doi.org/10.1016/j.scico.2012.07.014
33. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) ASE, pp. 164–173. ACM (2007). https://doi.org/10.1145/1321631.1321657
34. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Ramírez, A., Bilardi, G., Gschwind, M. (eds.) CF, pp. 43–54. ACM (2008). https://doi.org/10.1145/1366230.1366239
35. Yu, Y., Lin, Y., Hu, Z., Hidaka, S., Kato, H., Montrieux, L.: Maintaining invariant traceability through bidirectional transformations. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) ICSE, pp. 540–550. IEEE (2012). https://doi.org/10.1109/ICSE.2012.6227162

Dualizing Generalized Algebraic Data Types by Matrix Transposition

Klaus Ostermann(✉) and Julian Jabs

University of Tübingen, Tübingen, Germany
{klaus.ostermann,julian.jabs}@uni-tuebingen.de

Abstract. We characterize the relation between generalized algebraic datatypes (GADTs) with pattern matching on their constructors one hand, and generalized algebraic co-datatypes (GAcoDTs) with copattern matching on their destructors on the other hand: GADTs can be converted mechanically to GAcoDTs by refunctionalization, GAcoDTs can be converted mechanically to GADTs by defunctionalization, and both defunctionalization and refunctionalization correspond to a transposition of the matrix in which the equations for each constructor/destructor pair of the (co-)datatype are organized. We have defined a calculus, $GADT^T$, which unifies GADTs and GAcoDTs in such a way that GADTs and GAcoDTs are merely different ways to partition the program.

We have formalized the type system and operational semantics of $GADT^T$ in the Coq proof assistant and have mechanically verified the following results: (1) The type system of $GADT^T$ is sound, (2) defunctionalization and refunctionalization can translate GADTs to GAcoDTs and back, (3) both transformations are type- and semantics-preserving and are inverses of each other, (4) (co-)datatypes can be represented by matrices in such a way the aforementioned transformations correspond to matrix transposition, (5) GADTs are extensible in an exactly dual way to GAcoDTs; we thereby clarify folklore knowledge about the "expression problem".

We believe that the identification of this relationship can guide future language design of "dual features" for data and codata.

1 Introduction

The duality between data and codata, between construction and destruction, between smallest and largest fixed points, is a long-standing topic in the PL community. While some languages, such as Haskell, do not distinguish explicitly between data and codata, there has been a "growing consensus" [1] that the two should not be mixed up. Many ideas that are well-known from the data world have counterparts in the codata world. One work that is particularly relevant for this paper are copatterns, also proposed by Abel et al. [1]. Using copatterns,

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-3-319-89884-1_3) contains supplementary material, which is available to authorized users.

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 60–85, 2018.
https://doi.org/10.1007/978-3-319-89884-1_3

the language support for codata is very symmetrical to that for data: Data types are defined in terms of constructors, functions consuming data are defined using pattern matching on constructors; codata types are defined in terms of destructors, functions producing codata are defined using copattern matching on destructors.

Another example of designing dual features for codata is the recently proposed codata version of inductive data types [36]. However, coming up with these counterparts requires ingenuity. The overarching goal of this work is to replace the required ingenuity by a mechanical derivation. A key idea towards this goal has been proposed by Rendel et al. [31], namely to relate the data and codata worlds by refunctionalization [16] and defunctionalization [17,32].

Defunctionalization is a global program transformation to transform higher-order programs into first-order programs. By defunctionalizing a program, higher-order function types are replaced by sum types with one variant per function that exists in the program. For instance, if a program contains two functions of type $Nat \rightarrow Nat$, then these functions are represented by a sum type with two variants, one for each function, whereby the type components of each variant store the content of the free variables that show up in the function definition. Defunctionalized function calls become calls to a special first-order *apply* function which pattern-matches on the aforementioned sum type to dispatch the call to the right function body.

Refunctionalization is the inverse transformation, but traditionally it only works (easily) on programs that are in the image of defunctionalization [16]. In particular, it is not clear how to refunctionalize programs when there is more than one function (like *apply*) that pattern-matches on the same data type. Rendel et al. [31] have shown that this problem goes away when functions are generalized to arbitrary codata (with functions being the special codata type with only one *apply* destructor), because then every pattern-matching function in a program to be refunctionalized can be expressed as another destructor.

The main goal of this work is to extend the de- and refunctionalization correspondence between data and codata to generalized algebraic datatypes (GADTs) [8,40] and their codata counterpart, which we call Generalized Algebraic Codata types (GAcoDTs). More concretely, this paper makes the following contributions.

– We present the syntax, operational semantics, and type system of a language, $GADT^T$, that can express both GADTs and GAcoDTs. In this language, GADTs and GAcoDTs are unified in such a way that they are merely two different representations of an abstract "matrix" interface.
– We show that the type system is sound by proving progress and preservation [39].
– We formally define defunctionalization and refunctionalization, observe that they correspond to matrix transposition, and prove that GADTs and GAcoDTs are indistinguishable after hiding them behind the aforementioned matrix interface. We conclude that defunctionalization and refunctionalization preserve both operational semantics and typing.

– We prove that both GADTs and GAcoDTs can be extended in a modular way (with separate type checking) by "adding rows" to the corresponding matrix. Due to their matrix transposition relation, this means that the extensibility is exactly dual, which clarifies earlier informal results on the "expression problem" [11,33,37].
– The language and all results have been formalized and mechanically verified in the Coq proof assistant. The Coq sources are available in the supplemental material that accompanies this submission.
– As a small side contribution, if one considers only the GADT part of the language, this is to the best of our knowledge the first mechanically verified formalization of GADTs. It is also simpler than previous formalizations of GADTs because it is explicitly typed and hence avoids the complications of type inference.

The remainder of this paper is structured as follows. In Sect. 2 we give an informal overview of our main contributions by means of an example and using conventional concrete syntax. In Sect. 3 we present the syntax, operational semantics, and type system of $GADT^T$. Section 4 presents the aforementioned mechanically verified properties of $GADT^T$. In Sect. 5, we discuss applications and limitations of $GADT^T$, talk about termination/productivity and directions for future work, and describe how we formalized $GADT^T$ in Coq. Finally, Sect. 6 discusses related work and Sect. 7 concludes.

2 Informal Overview

Figure 1 illustrates the language design of $GADT^T$ in terms of an example. The left-hand side shows an example using GADTs and functions that pattern-match on GADT constructors. The right-hand side shows the same example using GAcoDTs and functions that copattern-match on GAcoDT destructors. The right-hand side is the refunctionalization of the left hand side; the left-hand side is the defunctionalization of the right-hand side.

Simply-Typed (Co)Datatypes. Let us first look at the `Nat` (co)datatype. Every data or codata type has an *arity*: The number of type arguments it receives. Since $GADT^T$ does only feature types of kind `*`, we simply state the number of type arguments in the (co)data type declaration. `Nat` receives zero type arguments, hence `Nat` illustrates the simply-typed setting with no type parameters. Functions in $GADT^T$, like `add` on the left-hand side, are first-order only; higher-order functions can be encoded as codata instead. Functions always (co)pattern-match on their first argument. (Co)pattern matching on multiple argument as well as nested and deep (co)pattern matching are not supported directly and must be encoded via auxiliary functions. We see that the refunctionalized version of `Nat` on the right-hand side turns constructors into functions, functions into destructors, and pattern matching into copattern matching. Abel et al. [1] use "dot notation" for copattern matching and destructor application; for instance, they

```
data Nat[0] where
  zero(): Nat
  succ(Nat): Nat

function add(Nat,Nat): Nat where
  add(zero(), x) = x
  add(succ(y),x) = succ(add(y,x))

data List[1] where
  nil[A]():  List[A]
  cons[A](A, List[A]): List[A]

function length[A](List[A]): Nat w..
  length[_](nil[_]) = 0
  length[B](cons[_](x,xs)) =
    succ(length[B](xs))

function sum(List[Nat]): Nat
  sum(nil[_]) = 0
  sum(cons[_](x,xs)) = x + sum(xs)

data Tree[1] where
  node(Nat): Tree[Nat]
  branch[A](List[Tree[A]])
            : Tree[List[A]]

function unwrap(Tree[Nat]): Nat w..
  unwrap(node(n)) = n
  unwrap(branch[_](xs)) = impossible

function width[A](Tree[A]): Nat w..
  width[_](node(n)) = 0
  width[_](branch[C](xs)) =
    length[C](xs)
```

```
codata Nat[0] where
  add(Nat,Nat) : Nat

function zero(): Nat where
  add(zero(),x) = x

function succ(Nat): Nat where
  add(succ(y),x) = succ(add(y,x))

codata List[1] where
  length[A](List[A]): Nat
  sum(List[Nat]): Nat

function nil[A](): List[A] where
  length[_](nil[_]) = 0
  sum(nil[_]) = 0

function cons[A](A, List[A]): List[A] w..
  length[B](cons[_](x,xs)) =
    succ(length[B](xs))
  sum(cons[_](x,xs)) = x + sum(xs)

codata Tree[1] where
  unwrap(Tree[Nat]) : Nat
  width[A](Tree[A]): Nat

function node(Nat): Tree[Nat] where
  unwrap(node(n)) = n
  width[_](node(n)) = 0

function branch[A](List[Tree[A]])
       : Tree [List[A]] where
  unwrap(branch[_](xs)) = impossible
  width[_](branch[C](xs)) =
    length[C](xs)
```

Fig. 1. The same example in the data fragment (left) and codata fragment (right)

List[1]	nil[A](): List[A]	cons[A](A, List[A]): List[A]
length[A](List[A]): Nat	length[_](nil[_]) = 0	length[B](cons[_](x,xs)) = succ(length[B](xs))
sum(List[Nat]): Nat	sum(nil[_]) = 0	sum(cons[_](x,xs)) = x + sum(xs)

Fig. 2. Matrix representation of List GADT from Fig. 1 (left)

List[1]	length[A](List[A]): Nat	sum(List[Nat]): Nat
nil[A](): List[A]	length[_](nil[_]) = 0	sum(nil[_]) = 0
cons[A](A, List[A]): List[A]	length[B](cons[_](x,xs)) = succ(length[B](xs))	sum(cons[_](x,xs)) = x + sum(xs)

Fig. 3. Matrix representation of List GAcoDT from Fig. 1 (right). This matrix is the transposition of Fig. 2.

would write `succ(y).add(x) = succ(y.add(x))` instead of `add(succ(y),x) = succ(add(y,x))` on the right-hand side of Fig. 1. We use the same syntax for constructor calls, function calls, and destructor calls because then the equations are not affected by de- and refunctionalization.

Parametric (Co)Datatypes. The `List` datatype illustrates the classical special case of GADTs with no indexing. Type arguments of constructors, functions, and destructors are both declared and passed via rectangular brackets `[...]` (loosely like in Scala). Like System F, $GADT^T$ has no type inference; all type annotations and type applications must be given explicitly. $GADT^T$ has a redundant way of binding type parameters. When defining an equation of a polymorphic function with a polymorphic first argument, we use square brackets to bind both the type parameters of the function and of the constructor/destructor on which we (co)pattern-match. For instance, in the equation `length[B](cons[_](x,xs)) = ...` on the left hand side, `B` is the type parameter of the `length` function, whereas the underscore (which we use if the type argument is not relevant, we could replace it by a proper type variable name) binds the type argument of the constructor with which the list was created. In this example, we could have also written the equation as `length[_](cons[B](x,xs)) = ...` because both type parameters must necessarily be the same, but in the general case we need access to both sets of type variables (as the next example will illustrate). It is important that we do not (co)pattern-match on type arguments, since this would destroy parametricity; rather, the `[...]` notation on the left hand side of an equation is only a binding construct for type variables.

Codatatypes also serve as a generalization of first-class functions. The code below shows how a definition of a general function type together with a specific family of first-class function `addn` (that can be passed as an argument and returned as a result), defined by a codata generator function with return type `Function[Nat,Nat]`.

```
codata Function[2] where
  apply[A,B](Function[A,B], A): B

function addn(Nat): Function[Nat,Nat] where
  apply(addn(n),m) = add(n,m)
```

Type Parameter Binding. Of those two sets of type parameter bindings, the one for functions is in a way always redundant because we could use the type variable declaration inside the function declaration instead. For instance, in the equation `length[B](cons[_](x,xs)) = succ(length[B](xs))` on the left hand side we could use the type parameter `A` of the enclosing function declaration instead. However, in $GADT^T$ the scope of the type variables in the function declaration does not extend to the equations and the type arguments must be bound anew in every equation. The reason for that is that we want to design the equations in such a way that they do not need to be touched when de/refunctionalizing a (co)datatype. For instance, when refunctionalizing a datatype, a function declaration is turned into a destructor declaration and what used to be a type argument that was bound in the enclosing function declaration becomes a type argument that is bound in a remote destructor declaration; to make type-checking modular we hence need a local binding construct. Our main goal in designing $GADT^T$ was not to make it convenient for programmers but to make the relation between GADTs and GAcoDTs as simple as possible; furthermore, a less verbose surface syntax could easily be added on top.

If we look at the corresponding `List` codatatype on the right-hand side, we see that the `sum` function from the left-hand side, which accepts only a list of numbers, turns into a destructor that is only applicable to those instances of `List` whose type parameter is `Nat`. This is similar to methods in object-oriented programming whose availability depends on type parameters [28], but here we see that this feature arises "mechanically" by the de/refunctionalization correspondence.

GA(co)DTs. The `Tree` (co)datatype illustrates a usage of GA(co)DTs that cannot be expressed with traditional parametric data types. We can see that by looking at the return type of the constructors of the `Tree` datatype; they are `Tree[Nat]` and `Tree[List[A]]` instead of `Tree[A]`. The `Tree` codatatype is also using the power of GAcoDTs in the `unwrap` destructor[1] because its first argument is different from `Tree[A]`. The GADT constructor `node(Nat): Tree[Nat]` turns into a function that returns a `Tree[Nat]` on the right hand side. The `Tree` example illustrates two additional issues that did not show up in the earlier examples.

First, it illustrates that type unification may make some pattern matches impossible, as illustrated by the `unwrap(branch[_](xs)) = impossible` equation on the left hand side. The equation is impossible, because the function argument type `Tree[Nat]` cannot be unified with the constructor return type `Tree[List[A]]`.[2] In $GADT^T$, we require that pattern matching is always complete, but impossible equations are not type-checked; the right-hand side can hence be filled with any dummy term. Second, the equation `width[_](branch[C](xs)) = length[C](xs)` illustrates the case where it is essential that we can bind constructor type arguments; otherwise we would have no name for the type argument we need to pass to `length`. Such type arguments are sometimes called *existential* or *phantom* [8] because if we have a branch of type `Tree[A]`, we only know that there exists some type that was used in the invocation of the `branch` constructor, but that type does not show up in the structure of `Tree[A]`.

We see again how both impossible equations and the need to access constructor type arguments translate naturally into corresponding features in the codata world. For impossible equations, we need to check whether the first destructor argument type can be unified with the function return type. Access to existential

[1] The `unwrap` destructor is meant to be used to extract the number from a tree that directly contains a number, i.e., a tree constructed with constructor `node`.

[2] This fits with our intention that `unwrap` should only work on a `node` (which directly contains a number).

constructor type arguments turns into access to local function types; conversely, access to existential destructor type arguments in the codata world turns into access to local function type arguments.

$GADT = GAcoDT^T$. We can see that the relation between GADTs and GAcoDTs is as promised when looking at Figs. 2 and 3. These two figures show a slightly different representation of the `List` (co)datatype and associated functions from Fig. 1. In this presentation, we have dropped all keywords from the language, such as `function`, `data` and `codata`. The reason for dropping these keywords is that now function signatures in the data fragment look the same as destructor signatures in the codata fragment, and constructor signatures in the data fragment look the same as function signatures in the codata fragment. Figure 2 organizes the datatype in the form of a matrix: the first row lists the datatype and its constructor signatures, the first column lists the signatures of the functions that pattern-match on the datatype, the inner cells represent the equations for each combination of constructor and function. Figure 3 does the same for the `List` codatatype: The first row lists the codatatype and its destructor signatures, the first column lists the signatures of functions that copattern-match on the codatatype, the inner cells represent the equations for each combination of function and destructor. We can now see that the relation between GADTs and GAcoDTs is now indeed rather simple: It is just matrix transposition.

An essential property of this transformation is that other (co)datatypes and functions are completely unaffected by the transformation. For instance, the `Tree` datatype (or codatatype, regardless of which version we use) looks the same, regardless of whether we encode `List` in data or in codata style. Defunctionalization and refunctionalization are still global transformations in that we need to find all functions that pattern-match on a datatype (for refunctionalization) or find all functions that copattern-match on a codatatype (for defunctionalization), but the rest of the program, including all clients of those (co)datatypes and functions, remain the same.

Infinite Codata, Termination, Productivity. The semantics of codata is usually defined via greatest fixed point constructions that include the possibility to represent "infinite" structures, such as streams. This is not the focus of this work, but since our examples so far did not feature such "infinite" structures but we do not want to give the impression that our codata types do somehow lack the expressiveness to express streams and the like, hence we show here an example of how to encode a stream of zeros, both in the codata representation (left) and, defunctionalized, in the data representation (right).

```
codata Stream where
  head(Stream) : Nat
  tail(Stream) : Stream

function zeros() : Stream
  head(zeros()) = zero()
  tail(zeros()) = zeros()
```

```
data Stream where
  zeros() : Stream

function head(Stream) : Nat
  head(zeros()) = zero()

function tail(Stream) : Stream
  tail(zeros()) = zeros()
```

Codata is also often associated with guarded corecursion to ensure productivity. In the copattern formulation of codata, productivity and termination coincide [2]. Due to our unified treatment of data and codata, a single check is sufficient for both termination/productivity of programs. In Sect. 5.3, we discuss a simple syntactic check that corresponds to both structural recursion and guarded corecursion.

Properties of $GADT^T$*.* In the remainder of this paper, we formalize $GADT^T$ in a style similar to the matrix representation of (co)datatypes we have just seen. We define typing rules and a small-step operational semantics and prove formal versions of the following informal theorems: (1) The type system of $GADT^T$ is sound (progress and preservation), (2) Defunctionalization and refunctionalization (that is, matrix transposition) of (co)datatypes preserves well-typedness and operational semantics, (3) Both types of matrices are modularly extensible in one dimension, namely by adding more rows to the matrix. This means that we can modularly add constructors or destructors and their respective equations without breaking type soundness as long as the new equations are sound themselves.

3 Formal Semantics

We have formalized $GADT^T$ and all associated theorems and proofs in Coq[3]. Here we present a traditional representation of the formal syntax using context-free grammars, a small-step operational semantics, and a type system.

We have formalized the language in such a way that we abstract over the physical representation of matrices as described in the previous section, hence we do not need to distinguish between GADTs and GAcoDTs. In the following, we say *constructor* to denote either a constructor of a datatype, or a function that copattern-matches on a codatatype. We say *destructor* to denote either a function that pattern-matches on a datatype, or a destructor of a codatatype. The language is defined in terms of constructors and destructors; we will later see that GADTs and GAcoDTs are merely different organizations of destructors and constructors.

3.1 Language Design Rationale

Our main goal in the formalization is to clarify the relation between GADTs and GAcoDTs, and not to design a calculus that is convenient to use as a

[3] Full Coq sources are available in the supplemental material.

programming language. Hence we have left out many standard features of programming calculi that would have made the description of that relation more complicated. In particular:

- Like System F, GADT^T requires explicit type annotations and explicit type application. Type inference could be added on top of the calculus, but this is not in the scope of this work.
- (Co)pattern matching is restricted in that every function must necessarily (co)pattern-match on its first argument, hence (co)pattern-matching on multiple arguments or "deep" (co)pattern matching must be encoded by auxiliary functions. Pattern matching is only supported for top-level function definitions; there is no "case" or "match" construct. Functions that are not supposed to (co)pattern-match (like the polymorphic identity function) must be encoded by a function that (co)pattern-matches on a dummy argument of type `Unit`.
- First-class functions are supported in the form of codata, but anonymous local first-class functions must be encoded via lambda lifting [3,25], that is, they must be encoded as top-level functions where the bindings for the free variables are passed as an extra parameter.
- Due to the abstraction over the physical representation of matrices we have not fixed the physical modular structure (a linearization of the matrix as text) of programs. Type checking of matrices simply iterates over all cells in an unspecified order. However, later on we will characterize GADTs and GAcoDTs as two physical renderings of matrices and formally prove the way in which those program organizations are extensible.

3.2 Notational Conventions

As usual, we use the same letters for both non-terminal symbols and meta-variables, e.g., t stands both for the non-terminal in the grammar for terms but inside inference rules it is a meta-variable that stands for any term. We use the notation $\overline{t}$ to denote a list $t_1, t_2, \ldots, t_{|\overline{t}|}$, where $|\overline{t}|$ is the length of the list. We also use list notation to denote iteration, e.g., $P, \Gamma \vdash \overline{t} : \overline{T}$ means $P, \Gamma \vdash t_1 : T_1, \ldots, P, \Gamma \vdash t_{|\overline{t}|} : T_{|\overline{t}|}$. To keep the notation readable, we write $\overline{x} : \overline{T}$ instead of $\overline{x : T}$ to denote $x_1 : T_1, \ldots, x_n : T_n$.

We use the notation $t[x := t']$ to denote the substitution of all free occurrences of x in t by t', and similarly $T[X := T']$ and $t[X := T']$ for the substitution of type variables in types and terms, respectively.

3.3 Syntax

The syntax of GADT^T is defined in Fig. 4. Types have the form $m[\overline{T}]$, where m is the name of a GADT or GAcoDT (in the following referred to as *matrix name*), and square brackets to denote type application. Types can contain type variables X. In the syntax of terms t, x denotes parameters that are bound by (co)pattern matching and y denotes other parameters. A constructor call $c[\overline{T}](\overline{t})$ takes zero or

$\boxed{\text{Syntax}}$

S, T	$::=$	$m[\overline{T}] \mid X$	*Types*
t	$::=$	$x \mid y \mid c[\overline{T}](\overline{t}) \mid d[\overline{T}](t, \overline{t})$	*Terms*
C	$::=$	$c[\overline{X}](\overline{T}) : m[\overline{T}]$	*Constructor Signature*
D	$::=$	$d[\overline{X}](m[\overline{T}], \overline{T}) : T$	*Destructor Signature*
e	$::=$	$d[\overline{Y}](c[\overline{X}](\overline{x}), \overline{y}) = t$	*Equations*
M	$=$	$(a, \gamma \in \overline{C}, \delta \in \overline{D}, \gamma \to \delta \to e)$	*Matrices*
P	$=$	$m \mapsto_{fin} M$	*Programs*
m	$\in$	Matrix names	
d	$\in$	Destructor names	
c	$\in$	Constructor names	
x	$\in$	Pattern Variable Names	
y	$\in$	Variable Names	
X, Y	$\in$	Type Variables	
a	$\in$	$\mathbb{N}$	*Arities*

$\boxed{\text{Operational Semantics} : P \vdash t \to t'}$

u, v	$::=$	$c[\overline{T}](\overline{v})$	*Values*
E	$::=$	$c[\overline{T}](\overline{v}, [], \overline{t}) \mid d[\overline{T}](\overline{v}, [], \overline{t})$	*Evaluation Context*

$$\frac{P \vdash t \to t'}{P \vdash E[t] \to E[t']} \qquad \text{(E-Ctx)}$$

$$\frac{\begin{array}{c} m \mapsto (a, \overline{C}, \overline{D}, \mathit{lookup}) \in P \\ D \in \overline{D} \qquad D = d[\ldots](m[\ldots], \ldots) \\ C \in \overline{C} \qquad C = c[\ldots](\ldots) \\ \mathit{lookup}(C, D) = d[\overline{Y}](c[\overline{X}](\overline{x}), \overline{y}) = t \end{array}}{P \vdash d[\overline{S}](c[\overline{T}](\overline{v}), \overline{u}) \to t[\overline{X} := \overline{S}, \overline{Y} := \overline{T}][\overline{x} := \overline{v}, \overline{y} := \overline{u}]} \qquad \text{(E-Fire)}$$

Fig. 4. Syntax and operational semantics of $GADT^T$

more arguments, whereas a destructor call $d[\overline{T}](t, \overline{t})$ takes at least one argument (namely the one to be destructed). Both destructors and constructors can have type parameters, which must be passed via square brackets.

A constructor signature $c[\overline{X}](\overline{T}) : m[\overline{T}]$ defines the number and types of parameters and the type parameters to the constructed type. Its output type cannot be a type variable but must be some concrete matrix type $m[\overline{T}]$. A destructor signature, on the other hand, must have a concrete matrix type as its first argument and can have an arbitrary return type. Equations $d[\overline{Y}](c[\overline{X}](\overline{x}), \overline{y}) = t$ define what happens when a constructor c meets a destructor d. The $\overline{x}$ bind the components of the constructor, whereas the $\overline{y}$ bind the remaining parameters of the destructor call. We also bind both the type arguments to the constructor $\overline{X}$

and the destructor $\overline{Y}$, such that they can be used inside t. In many cases, the $\overline{X}$ will provide access to the same types as $\overline{Y}$, but in the general case we need both because both constructors and destructors may contain phantom types [8].

Matrices M are an abstract representation of both GADTs and GAcoDTs, together with the functions that pattern-match (for GADTs) or copattern-match (for GAcoDTs) on the GA(co)DTs. A matrix has an arity a (the number of type parameters it receives), a list of constructors γ, and a list of destructors δ. It also has a lookup function that returns an equation for every constructor/destructor pair on which the matrix is defined (hence the type of matrices is a dependent type). There must be an equation for each constructor/destructor pair, but in the case of impossible combinations, the equations are not type-checked and some dummy term can be inserted. A program P is just a finite mapping from matrix names to matrices.

3.4 Operational Semantics

We define the operational semantics, also in Fig. 4, via an evaluation context E, which, together with E-Ctx, defines a standard call-by-value left-to-right evaluation order. Not surprisingly, the only interesting rule is E-Fire, which defines the reduction behavior when a destructor meets a constructor. We look up the corresponding matrix in the program and look up the equation for that constructor/destructor pair. In the body of the equation, t, we perform two substitutions: (1) We substitute the formal type arguments $\overline{X}$ and $\overline{Y}$ by the current type arguments $\overline{S}$ and $\overline{T}$, and (2) we substitute the pattern variables $\overline{x}$ by the components $\overline{v}$ of the constructor and the variables $\overline{y}$ by the current arguments $\overline{u}$.

3.5 Typing

The typing and well-formedness rules are defined in Fig. 5. Let us first look at the typing of terms. The rules for variable lookup are standard. The constructor rule T-Const checks that the number of type- and term arguments matches the declaration and checks the type of all arguments, whereby the type variables are substituted by the type arguments of the actual constructor call. Constructor names must be globally unique, hence the matrix to which the constructor belongs is not relevant.

This is different for typing destructor calls (T-Dest). A destructor is resolved by first determining the matrix m of the first destructor argument, and then the destructor is looked up in that matrix. It is hence OK if the same destructor name shows up in multiple matrices. When considering codata as “objects” like in object-oriented programming [24], this corresponds to the familiar situation that different classes can define methods with the same name. In the GADT case, this corresponds to allowing multiple pattern-matching functions of the same name that are disambiguated by the type of their first argument.

In Wf-Eq, we construct the appropriate typing context to type-check the right hand side of equations. We allow implicit α-renaming of type variables

$\boxed{\text{Term Typing} : P, \Gamma \vdash t : T}$

$$\Gamma ::= \epsilon \mid x : T, \Gamma \mid y : T, \Gamma \quad \textit{Typing Contexts}$$

$$\frac{x : T \in \Gamma}{P, \Gamma \vdash x : T}\ (\text{T-PVar}) \qquad \frac{y : T \in \Gamma}{P, \Gamma \vdash y : T}\ (\text{T-Var})$$

$$\frac{\begin{array}{c} P, \Gamma \vdash t : m[\overline{T}][\overline{X} := \overline{S}] \\ m \mapsto (\ldots, \ldots d[\overline{X}](m[\overline{T}], \overline{U}) : T \ldots, \ldots) \in P \\ \forall i. P, \Gamma \vdash t_i : U_i[\overline{X} := \overline{S}] \\ |\overline{X}| = |\overline{S}| \qquad |\overline{U}| = |\overline{t}| \end{array}}{P, \Gamma \vdash d[\overline{S}](t, \overline{t}) : T[\overline{X} := \overline{S}]}\ (\text{T-Dest})$$

$$\frac{\begin{array}{c} \ldots \mapsto (\ldots c[\overline{X}](\overline{T}) : T \ldots, \ldots, \ldots) \in P \\ \forall i. P, \Gamma \vdash t_i : T_i[\overline{X} := \overline{S}] \\ |\overline{X}| = |\overline{S}| \qquad |\overline{T}| = |\overline{t}| \end{array}}{P, \Gamma \vdash c[\overline{S}](\overline{t}) : T[\overline{X} := \overline{S}]}\ (\text{T-Const})$$

$\boxed{\text{Well-Formedness}}$

$$\frac{\begin{array}{c} C = c[\overline{X'}](\overline{T}) : m[\overline{S}] \qquad |\overline{X}| = |\overline{X'}| \\ D = d[\overline{Y'}](m[\overline{S'}], \overline{T'}) : T \qquad |\overline{Y}| = |\overline{Y'}| \\ \textit{all-distinct}(\overline{X}, \overline{Y}) \qquad \textit{all-distinct}(\overline{X'}, \overline{Y'}) \\ \textit{most-general-unifier}(m[\overline{S}], m[\overline{S'}]) = \sigma \\ P, \overline{x} : \sigma(\overline{T}), \overline{y} : \sigma(\overline{T'}) \vdash \sigma(t[\overline{X} := \overline{X'}, \overline{Y} := \overline{Y'}]) : \sigma(T) \end{array}}{P, m \vdash d[\overline{Y}](c[\overline{X}](\overline{x}), \overline{y}) = t \text{ OK in } C, D}\ (\text{Wf-Eq})$$

$$\frac{\begin{array}{c} C = \ldots : m[\overline{S}] \qquad D = \ldots (m[\overline{S'}], \ldots) : \ldots \\ \textit{most-general-unifier}(m[\overline{S}], m[\overline{S'}]) = \text{error} \end{array}}{P, m \vdash d[\overline{Y}](c[\overline{X}](\overline{x}), \overline{y}) = t \text{ OK in } C, D}\ (\text{Wf-Infsble})$$

$$\frac{|\overline{S}| = a \qquad FV(\overline{T}) \subseteq \overline{X} \qquad FV(\overline{S}) \subseteq \overline{X}}{c[\overline{X}](\overline{T}) : m[\overline{S}] \text{ OK in } m, a}\ (\text{Wf-Constr})$$

$$\frac{|\overline{S}| = a \qquad FV(\overline{S}) \subseteq \overline{Y} \qquad FV(\overline{T}) \subseteq \overline{Y}}{d[\overline{Y}](m[\overline{S}], \overline{T}) : T \text{ OK in } m, a}\ (\text{Wf-Destr})$$

$$\frac{\begin{array}{c} \forall C \in \overline{C}, \forall D \in \overline{D}, \\ C \text{ OK in } m, a \\ D \text{ OK in } m, a \\ P, m \vdash \textit{lookup}(C, D) \text{ OK in } C, D \\ \textit{all-names-distinct}(\overline{D}) \end{array}}{m \mapsto (a, \overline{C}, \overline{D}, \textit{lookup}) \text{ OK in } P}\ (\text{Wf-Matr})$$

$$\frac{\begin{array}{c} \forall m \in \textit{dom}(P), m \mapsto P(m) \text{ OK in } P \\ \textit{all-names-distinct}(\textit{ctors}(P)) \end{array}}{P \text{ OK}}\ (\text{Wf-Prog})$$

Fig. 5. Typing and well-formedness

to prevent accidental name clashes (checked by *all-distinct*). We compute the most general unifier of the two matrix types in the constructor and destructor, respectively, to combine the type knowledge about the matrix type from the constructor and destructor type. If no such unifier exists, the equation is vacuously well-formed because the particular combination of constructor and destructor can never occur during execution of well-typed terms (Wf-Infsble). Otherwise, we use the unifier σ and apply it to the given type annotations to type-check the term t. A unifier σ is a mapping from type variables to types, but we also use the notation $\sigma(t)$ and $\sigma(T)$ to apply σ to all occurrences of type variables inside a term t or a type T, respectively.

Constructor and destructor signatures are well-formed if they apply the correct number of type parameters to the matrix type and contain no free type variables (Wf-Constr and Wf-Destr). A matrix is type-checked by making sure that all constructor and destructor signatures are well-formed, that all equations are well-formed for every constructor/destructor combination, and that destructor names are unique in the matrix (Wf-Matr). To check uniqueness of names, we use *all-names-distinct*, which checks for a given list of signatures that all of their names are distinct. A program is well-formed if all of its matrices typecheck and the constructor signatures of the program (retrieved by *ctors*) are globally unique (Wf-Prog).

3.6 GADTs and GAcoDTs

In the formalization so far, we have deliberately kept matrices abstract as a kind of abstract data type. Now we can bring in the harvest of our language design. GADTs and GAcoDTs are two different physical representations of matrices, see Fig. 6. They both contain nested vectors of equations and differ only in the order of the indices. With GADTs, the column labels are constructors and the row labels functions and a row corresponds to a function defined by pattern matching, with one equation for each case of the GADT. With GAcoDTs, the column labels are destructors, the row labels are functions, and a row corresponds to a function defined by copattern matching, with one equation for each case of

$$\begin{array}{lcl}
M_{GADT} & = & (a, \gamma \in \overline{C}, \delta \in \overline{D}, \{e_{D,C} | D \in \delta, C \in \gamma\}) \\
M_{GAcoDT} & = & (a, \gamma \in \overline{C}, \delta \in \overline{D}, \{e_{C,D} | C \in \gamma, D \in \delta\}) \\
\\
\mathit{mkmatrix} & : & M_{GADT} + M_{GAcoDT} \rightarrow M \\
\mathit{mkmatrix} & = & \text{— obvious; omitted} \\
\\
\mathit{refunctionalize} & : & M_{GADT} \rightarrow M_{GAcoDT} \\
\mathit{refunctionalize} & = & \mathit{transpose} \\
\\
\mathit{defunctionalize} & : & M_{GAcoDT} \rightarrow M_{GADT} \\
\mathit{defunctionalize} & = & \mathit{transpose}
\end{array}$$

Fig. 6. GADTs and GAcoDTs

the GAcoDT. Hence both *defunctionalize* and *refunctionalize*, which swap the respective organization of the matrix, are just matrix transposition.

4 Properties of $\boldsymbol{GADT^T}$

In this section, we prove type soundness for $GADT^T$, the preservation of typing and operational semantics under de- and refunctionalization, and that our physical matrix representations of GADTs and GAcoDTs are accurate with respect to extension. All of these properties have been formalized and proven in Coq, based upon our Coq formalization of the previous section's formal syntax, semantics, and type system.

4.1 Type Soundness

We start with the usual progress and preservation theorems.

Theorem 1 (Progress). *If P is a well-formed program and t is a term with no free type variables and $P, \epsilon \vdash t : T$, then t is either a value v, or there exists a term t' such that $P \vdash t \rightarrow t'$.*

The proof of this theorem is a simple induction proof using a standard canonical forms lemma [30].

Preservation is much harder to prove. Often, preservation is proved using a substitution lemma which states that the substitution of a (term) variable by a term of the same type does not change the type of terms containing that term variable [30]. In $GADT^T$, this lemma looks as follows:

Lemma 1 (Term Substitution). *If $\overline{t}$ is a list of terms with $P, \epsilon \vdash \overline{t} : \overline{T}$ and $\overline{t'}$ is a list of terms with $P, \epsilon \vdash \overline{t'} : \overline{T'}$ and t is a term with $P, \overline{x} : \overline{T}, \overline{y} : \overline{T'} \vdash t : T$, then $P, \epsilon \vdash t[\overline{x} := \overline{t}, \overline{y} := \overline{t'}] : T$*

However, in E-Fire we perform both a substitution of terms and of types, hence the term substitution lemma is not enough to prove preservation; we also need a type substitution lemma.

Lemma 2 (Type Substitution). *If $P, \Gamma \vdash t : T$, then $P, \Gamma[\overline{X} := \overline{T}] \vdash t[\overline{X} := \overline{T}] : T[\overline{X} := \overline{T}]$*

The proof of this lemma requires various auxiliary lemmas about properties (such as associativity) of type substitution. Taken together, these two lemmas are the two main intermediate results to prove the desired preservation theorem.

Theorem 2 (Preservation). *If P is a well-formed program and t is a term with no free type variables and $P, \epsilon \vdash t : T$ and $P \vdash t \rightarrow t'$, then $P, \epsilon \vdash t' : T$.*

4.2 Defunctionalization and Refunctionalization

The preservation of typing and operational semantics by de/refunctionalization is a trivial consequence of the lemma below, which holds due to the fact that both de- and refunctionalization is merely matrix transposition, see Fig. 6, and that the embedding *mkmatrix* of the physical matrices into the abstract representation ignores the organization of the physical matrices.

Lemma 3 (Matrix Transposition)
$\forall m \in M_{GADT}$, $\mathit{mkmatrix}(m) = \mathit{mkmatrix}(\mathit{refunctionalize}(m))$.
$\forall m \in M_{GAcoDT}$, $\mathit{mkmatrix}(m) = \mathit{mkmatrix}(\mathit{defunctionalize}(m))$.

Corollary 1 (Preservation of typing and reduction). *De/refunctionalization of a matrix does not change the well-typedness of a program or the operational semantics of a term.*

4.3 Extensibility

So far, we have seen that our chosen physical matrix representations are amenable to easy proofs of the preservation of properties under de- and refunctionalization. However, are they also indeed accurate representations of GADTs and GAcoDTs? GADTs and GAcoDTs are utilized due to their *extensibility* along the destructor or constructor dimension, respectively, so we want this to be reflected by our representations.

We assume that matrices are represented as a traditional linear program by reading them row-by-row. Adding a new row is a non-invasive operation (adding to the program), whereas adding a column requires changes to the existing program.

We want to be able to extend our matrix representations with a new row, respectively representing the addition of a new destructor or constructor, without breaking well-typedness as long as the *newly added* equations typecheck with respect to the complete new program, and uniqueness of destructor/constructor names is preserved (globally, in the constructor case)[4].

In order to formally state that this is indeed the case, we first formally capture extension of GADT and GAcoDT matrices with the following definitions. These already include the preservation of local uniqueness as a condition, i.e., the name of the newly added destructor or constructor must be fresh within the matrix.

Definition 1 (GADT extension). *Consider an* $m \in M_{GADT}$ *with* $m = (a, \gamma, \delta, \{e_{D,C} | D \in \delta, C \in \gamma\})$. *For any* $D' \in \overline{D}, D' \notin \delta$, *and equations* $e_{D',C}$, *for each* $C \in \gamma$, *we call* $(a, \gamma, \delta \cup \{D'\}, \{e_{D,C} | D \in \delta \cup \{D'\}, C \in \gamma\})$ *a* GADT extension of m *with* D' *and* $\{e_{D',C} | C \in \gamma\}$.

[4] The counterpart to this property on the side of the operational semantics is that the reduction relation of the new program restricted to terms befitting the old program equals the reduction relation of the old program; this however we omitted as it holds trivially when uniqueness is preserved.

Definition 2 (GAcoDT extension). *Consider an* $m \in M_{GAcoDT}$ *with* $m = (a, \gamma, \delta, \{e_{C,D} | C \in \gamma, D \in \delta\})$. *For any* $C' \in \overline{C}, C' \notin \gamma$, *and equations* $e_{C',D}$, *for each* $D \in \delta$, *we call* $(a, \gamma \cup \{C'\}, \delta, \{e_{C,D} | C \in \gamma \cup \{C'\}, D \in \delta\})$ *a* GAcoDT extension of m *with* C' *and* $\{e_{C',D} | D \in \delta\}$.

We now straightforwardly lift these definitions to programs: A program P' *is a GA(co)DT extension (with some signature and equations) of another program* P if their matrices are identical except for one matrix name, and the underlying physical matrix (packed with *mkmatrix*) assigned to this name under P' is GA(co)DT extension (with this signature and equations) of the underlying physical matrix assigned under P.

Using this terminology we can now formally state and prove the extensibility of GADTs and GAcoDTs:

Theorem 3 (Datatype Extensibility). *If* P *is a well-formed program, and* P' *is a GADT extension of* P *with* D' *and equations* $\{e_{D',C} | C \in \gamma\}$, *for the constructor signatures* γ *of the matrix to be extended, such that* $P', m \vdash e_{D',C}$ *OK in C,D' for each* $C \in \gamma$, *then* P' *is well-formed.*

Theorem 4 (Codatatype Extensibility). *If* P *is a well-formed program, and* P' *is a GAcoDT extension of* P *with* C', *where the name of* C' *is different from all constructor names in* P, *and equations* $\{e_{C',D} | D \in \delta\}$, *for the destructor signatures* δ *of the matrix to be extended, such that* $P', m \vdash e_{C',D}$ *OK in C',D for each* $D \in \delta$, *then* P' *is well-formed.*

In other words, in both cases we can type-check each row of a matrix in isolation, and if we put those rows together the resulting matrix and program containing that matrix will be well-formed. The results justify the familiar physical representation of programs where the variants of a GADT are fixed but we can freely add new functions that pattern-match on that GADT (and correspondingly for GAcoDTs).

5 Discussion

In this section we discuss applications and limitations of our work, talk about directions for future work, and describe the Coq formalization of the definitions and proofs.

5.1 Applications

Language Design. The most obvious application of our approach is to guide programming language design, namely by designing its features in such a way that the correspondence by de/refunctionalization is preserved. We believe that we can find "gaps" in existing languages by checking whether the corresponding dual feature exists, or massaging the language feature in such a way that a clear dual exists. For instance, on the datatype and pattern matching side, many features exist that have no clear counterpart on the codata side yet, such as pattern matching on multiple arguments, non-linear pattern matching, or pattern guards [22]. Some vaguely dual features exist on the codata side understood as "objects", e.g. in the form of multi dispatch (such as [10]) or predicate dispatch [21]. We believe that the relation between pattern matching on multiple arguments and multi dispatch is a particularly interesting direction for future work, since it would entail generalizing our two-dimensional matrices to matrices of arbitrary dimension.

Arguably, codata is the essence of object-oriented programming [12]. In any case, we believe that our design can also help to design object-oriented language features. For instance, there has been previous works on "object-oriented" GADTs [20,26] using extensions of generic types with certain classes of constraints. For instance, in Kennedy and Russo's [26] work, a list interface could be defined like this:

```
interface List<A> {
  Integer size();
  Integer sum() where A=Integer; // Kennedy & Russo's syntax
}
```

If we compare this interface with the `List` codata type in Fig. 1 (right hand side), then we can see that such constraints are readily supported by GAcoDTs; not because this feature was explicitly added but because it arises mechanically from dualizing GADTs.

As another potential influence on language design, we believe that "closedness" under defunctionalization and refunctionalization can be a desirable language design quality that prevents oddities that things can be expressed better using codata than using data (or vice versa). For instance, Carette et al. [5] propose a program representation (basically again a form of Church encoding, hence a codata encoding) that works in a simple Haskell'98 language but whose datatype representation would require GADTs. This suggests a language design flaw in that the codata fragment of functions supports a more powerful type system than the data fragment of (non-generalized) algebraic data types. That is, the type arguments of a codata generator function's result type may be arbitrarily specialized, e.g., the result type might be `List[Nat]`, while the type of a constructor must be fully generic, e.g., `List[A]`. Our approach gives a criterion on when the type systems for both sides are "in sync".

De/Refunctionalization as a Programmer Tool. Semantics-preserving program transformations are not only interesting on the meta-level of programming language design but also because they define an equivalence relation on programs. For instance, consider the program on the left-hand side of Fig. 7, written in our GAcoDT language. `Nat` is a representation of Church-encoded[5] natural numbers as a GAcoDT with arity zero and a singular destructor `fold` with a type

[5] This form of typed Church encoding is sometimes called Böhm-Berarducci encoding [4].

```
codata Func[2] where
  apply[A,B](Func[A,B], A) : B

codata Nat[0] where
  fold[A](Nat,A,Func[A,A]) : A

fun zero(): Nat where
  fold[A](zero(),z,s) = z

fun succ(Nat): Nat where
  fold[A](succ(n),z,s) =
     apply[A,A](s,fold[A](n,z,s))
```

```
data Nat[0] where
  zero() : Nat
  succ(Nat) : Nat

fun fold[A](Nat,A,Func[A,A]) : A where
  fold[A](zero(),z,s) = z
  fold[A](succ(n),z,s) =
     apply[A,A](s, fold[A](n,z,s))
```

Fig. 7. Defunctionalizing Church-encoded numbers (left) yields Peano numbers with a fold function (right)

parameter `A`. Defunctionalizing `Nat` yields the familiar Peano numbers with the standard fold function (right-hand side).

Such equivalences have been identified as being useful to identify different forms of programs that are "the same elephant". For instance, Olivier Danvy and associates [16,17] have used defunctionalization, refunctionalization, and some other transformations such as CPS-transformation to inter-derive "semantic artifacts" such as big-step semantics, small-step semantics, and abstract machines ("The inter-derivations illustrated here witness a striking unity of computation, be this for reduction semantics, abstract machines, and normalization function: they all truly define the same elephant." – Danvy et al. [15]).

The applicability of these transformations is widened by our approach since we support arbitrary codata and not just functions. Exploring these new possibilities is an interesting area of future work.

Furthermore, programmers can employ our transformation as a tool for a more practical purpose. Consider that at some point during the development of a large software, it might have been determined that the extensibility dimension for a particular aspect should be switched. That is, it is now thought that instead of allowing to add new variants (constructors), the software would be better poised by fixing the variants and allowing the addition of new operations (destructors), or vice versa. In the case that at this point it is further possible to make a closed-world assumption with regards to the particular type (represented as a matrix), since clients of the code are known and can be dealt with, it might seem reasonable to transpose the matrix representing that type. With $GADT^T$, it is possible to do this independently of the other matrices in the program. (As already discussed, $GADT^T$ in its present form doesn't aim to be particularly developer-friendly, but we expect further language layers to be placed on top of $GADT^T$ to remedy this eventually.)

Compiler Optimizations. To be able to use our automatizable transformation as a programmer tool, it was important to be able to make a closed-world assumption, where we have the entire program, or more precisely, the part which involves the matrix under consideration, at our disposal. A more automated process where such a kind of assumption can often be readily made is compilation. There, our matrix transposition transformation can be employed for a whole program optimization (such as [6]), as follows. An opportunity for optimization presents itself to the compiler when it is basically able to recognize an abstract machine in the code; optimizing this abstract machine is then an intermediate step, more generally applicable, that precedes hardware-specific optimizations [18]. As outlined above, defunctionalization can turn higher-order programs into first-order programs where this machine might be apparent. With our pair of languages, using our readily automatizable defunctionalization (matrix transposition), it is possible to turn GAcoDT code into GADT code during the compilation phase. Then the compiler can leverage the potentially recognizable abstract machine form of the GADT code for its optimizations.

5.2 Limitations

As we said, our design rationale for $GADT^T$ was to clarify the relation between GADTs and GAcoDTs, not to provide a convenient language for developers. Here we discuss some ways to address the limitations resulting from that decision.

Local (Co)Pattern Matching, Including λ. A significant limitation of $GADT^T$ is that (co)pattern matching is only allowed on the top-level; we don't have "case" (or "match") constructs on the term level. Any local (co)pattern matching, however, can be converted to the top-level form by extracting it to a new top-level function definition. Variables free within the (co)pattern matching term must be passed to this function as arguments. In particular, anonymous local first-class functions, i.e., λ expressions, are a form of local copattern matching which can be encoded in this way; this particular conversion is traditionally called lambda lifting.

(Co)Pattern Matching on Zero or More Arguments. (Co)pattern matching in $GADT^T$ is only possible on a single, distinguished argument (in our presentation, the first, but this is not important). Nested and multiple-argument matching can be encoded by *unnesting* à la Setzer et al. [35], producing auxiliary functions.

In $GADT^T$, it is further not possible to define a function without any (co) pattern matching entirely. The workaround of (co)pattern matching on a dummy argument of type `Unit` is simple, but it is not obvious how to reconcile this encoding with the symmetry of de/refunctionalization.

Type Inference. We have deliberately avoided the question of type inference in this work. In general, we expect that the ample existing works on type inference for GADTs (such as Peyton Jones et al. [29], Schrijvers et al. [34], Chen and Erwig [7]) can be adapted to our setting and will also work for GAcoDTs. We see one complication, though: Due to the fact that destructors are only locally unique in $GADT^T$, the (co)datatype the destructor belongs to must first be found via the type inferred for its distinguished, destructed argument. In other words, we do not know which destructor signature to consider before we know the destructed argument's type. This means that a type inference system which works inwards only, i.e., it discovers the types of the destructor arguments by looking at the signature, possibly leaving unification variables, and then checks that the recursively discovered types for the arguments conform, will not work.

5.3 Termination and Productivity

While termination and productivity are not in the focus of this paper, we want to mention that our unified treatment of data and codata can also lead to a unified treatment of termination and productivity.

Here we want to illustrate informally that a simple syntactic criterion is sufficient to allow structural recursion and guarded corecursion. Syntactic termination checks are not expressive enough for many situations, hence we leave a proper treatment of termination/productivity checking (such as with sized types [2]) for future work; the purpose of this discussion is merely to illustrate that termination checking could also benefit from unified data and codata and not to propose a practically useful termination checker.

The basic idea is to restrict destructor calls in the right-hand sides of equations to have the form $d[\overline{T}](x, \overline{t})$ instead of $d[\overline{T}](t, \overline{t})$. That is to say, in destructor calls, we only allow variables from *within* the constructor pattern of the left-hand side. This criterion already guarantees termination (and hence also productivity [2]) in our system, i.e. the finiteness of all reduction sequences, which can be shown with the usual argument of a property that strictly decreases under reduction. A reduction step in $GADT^T$ with right-hand sides restricted like that strictly decreases, under lexicographic order, the pair of

1. the maximum of all the first (destructed) arguments depths in destructor calls of the term, and
2. the sequence which counts how often each destructed argument depth appears in the term, starting with the maximum depth and going downward; those sequences are themselves lexicographically ordered.

This strict decrease can be proved by induction on the derivation of the reduction step. Since there are no infinitely decreasing sequences of these pairs, any reduction sequence must be finite. Note that our criterion in itself excludes far too many programs to be anywhere near practical, but it is readily conceivable how to relax it to only *recursive* calls together with a check that excludes mutual recursion.[6]

Let's look at Fig. 7 once more to illustrate that this criterion corresponds to both structural recursion and guarded corecursion. In the right-hand side of Fig. 7 we see that the first argument to the recursive call in the last line is `n`, which is allowed by our restriction because it is a syntactic part of the original input,

[6] For instance one might request the programmer to order the destructor names such that in equations for a certain destructor only destructors of lower order may be called.

`succ(n)` (structural recursion). The call to `apply` is not a problem because it is not a recursive call.[7] At the same time, if we look at the last line in the left-hand side of Fig. 7, we see that the criterion also corresponds to guarded corecursion. With copatterns, guarded corecursion means that we do not destruct the result of a recursive call (the "guard" itself is implicit in the pattern on the left-hand side of the equation). However, destructing that result would mean that we would have to call a destructor with the recursive call as its first argument, which is again forbidden by the syntactic criterion.

5.4 Going Beyond System F-like Polymorphism

A particularly interesting direction for future work is to extend $GADT^T$ and go beyond the System F-like polymorphism. For instance, F_ω contains a copy of the simply-typed lambda calculus on the type level. Could one also generalize type-level functions to arbitrary codata and maybe use a variant of $GADT^T$ on the type level? Can dependent products like in the calculus of constructions [13] be generalized in a similar way? Can inductive types like in the calculus of inductive constructions be formulated such that there is a dual that is also related by de/refunctionalization? Thibodeau et al. [36] have formulated such a dual, but whether it can be massaged to fit into the setting described here is not obvious.

5.5 Coq Formalization

Our Coq formalization is quite close to the traditional presentation chosen for this paper, but there are some technical differences. Both term and type variables are encoded via de Bruijn indices, which is rather standard for programming language mechanization. More interestingly, the syntax of the language in the Coq formalization expresses some of the constraints we express here via typing rules instead via dependent types. Specifically, terms and types are indexed by the type variables that can appear inside. To represent matrices, we have developed a small library of dependently typed tables (where the cell types can depend on the row and column labels), such that the matrix type already guarantees that all type variables that show up in terms and types are bound. An earlier version of the formalization and the soundness proof used explicit well-formedness constraints to guarantee that all type variables are bound; the type soundness proof for this version was about twice as long as the one using dependent types. On the flip side, we had to "pay" for using the dependent types in the form of many annoying "type casts" in definitions and theorems owing to the fact that Coq's equality is intensional and not extensional [9, Sect. 10.3]. Finally, instead of using an evaluation context to define evaluation order like we did in Fig. 4, we have used traditional congruence rules. In the reduction relation as formalized in Coq, a single step can actually correspond to multiple steps in the formalization presented in the paper; however, this is just a minor technicality to slightly simplify the proofs.

[7] As long as we avoid mutual recursion, for instance by ensuring `fold` > `apply`.

6 Related Work

"Theoreticians appreciate duality because it reveals deep symmetries. Practitioners appreciate duality because it offers two-for-the-price-of-one economy." This quote from Wadler [38] describes the spirit behind the design of $GADT^T$, but of course this is not the first paper to talk about duality in programming languages. We have already discussed the most closely related works in previous sections; here, we compare $GADT^T$ with theoretical calculi with related duality properties and point out an aspect of practical programming for which the duality of $GADT^T$ is relevant.

Codata. Hagino [23] pioneered the idea of dualizing data types: Whereas data types are used to define a type by the ways to *construct* it, codatatypes are dual to them in the sense that they are specified by their *deconstructions*. Abel et al. [1] introduce copatterns which allow functions producing codata to be defined by matching on the destructors of the result codatatype, dually to matching on the constructors of the argument datatype. All these developments occur in a world where function types are a given. The symmetric codata and data language fragments proposed by Rendel et al. [31] deviate from this: By enhancing destructor signatures with argument types, they provide a form of codata that is a generalization of first-class functions. Both the works by Rendel et al. [31] and Abel et al. [1] are simply-typed.

The (co)datatypes in the calculus of ownen and Ariola [19] also allow for user-defined function types. Their focus is different from ours, though, as they are mostly interested in evaluation strategies and their duality, and with regards to their calculus itself they work in an untyped setting. What is interesting in comparison with $GADT^T$ is how their (co)datatype declarations and signatures are inherently more symmetric as they essentially describe a type system for the parametric sequent calculus. As such, the position of additional arguments in the destructor signatures has a mirror counterparts in constructor signatures (to highlight this, Downen and Ariola [19] refer to destructors as "co-constructors").

Duality of Computations and Values. Staying on with the idea of avoiding function types as primitives for a moment, Wadler [38] presents a "dual calculus" in which the previously astonishing result that call-by-name is De Morgan-dual to call-by-value [14] is clarified by defining implication (corresponding to function types via the Curry-Howard isomorphism) in two different ways dependent on the intended corresponding evaluation regime. A somewhat similar approach, but perhaps more directly related to the data/codata duality, that also deals with the "troubling" coexistence of call-by-value and call-by-name, was proposed by Levy [27]. Levy [27] presents a calculus with a new evaluation regime, *call-by-push value* (CBPV), which subsumes call-by-value and call-by-name by encoding the local choice for either in the terms of the calculus. More specifically, there are two kinds of terms in the CBPV calculus: computations and values, which can be inter-converted by "thunking" and "forcing". The terms for computations and values are said to be of *positive* type and of *negative* type, respectively. Thibodeau et al. [36] have built their calculus, which extends codatatypes to indexed codatatypes, on top of CBPV, with datatypes being positive and codatatypes being negative. We think that, when extending $GADT^T$ with local (co)pattern matching on the term level, perhaps with pattern and copattern matching terms mixed, it might be helpful to similarly recast the resulting language as a modification of the CBPV calculus of Levy [27].

7 Conclusions

We have presented a formal calculus, $GADT^T$, which uniformly describes both GADTs and their dual, GAcoDTs. GADTs and GAcoDTs can be converted back and forth by defunctionalization and refunctionalization, both of which correspond to a transposition of the matrix of the equations for each pair of constructor/destructor. We have formalized the calculus in Coq and mechanically verified its type soundness, its extensibility properties, and the preservation of typing and operational semantics by defunctionalization and refunctionalization.

We believe that our work can be of help for future language design since it describes a methodology to get a kind of "sweet spot" where data and codata constructs (including functions) are "in sync". We think that it can also be useful as a general program transformation tool, both on the program level as a kind of refactoring tool, but also as part of compilers and runtime systems. Finally, since codata is quite related to objects in object-oriented programming, we hope that our approach can help to clarify their relation and design languages which subsume both traditional functional and object-oriented languages.

Acknowledgments. We would like to thank Tillmann Rendel and Julia Trieflinger for providing some early ideas for the design of what eventually became $GADT^T$. This work was supported by DFG project OS 293/3-1.

References

1. Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: programming infinite structures by observations. In: Proceedings of the Symposium on Principles of Programming Languages, pp. 27–38. ACM (2013)
2. Abel, A.M., Pientka, B.: Wellfounded recursion with copatterns: a unified approach to termination and productivity. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, pp. 185–196. ACM, New York (2013)
3. Augustsson, L.: A compiler for lazy ML. In: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, LFP 1984, pp. 218–227. ACM, New York (1984)
4. Böhm, C., Berarducci, A.: Automatic synthesis of typed lambda-programs on term algebras. Theor. Comput. Sci. **39**, 135–154 (1985)
5. Carette, J., Kiselyov, O., Shan, C.: Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(5), 509–543 (2009)

6. Chambers, C., Dean, J., Grove, D.: Whole-program optimization of object-oriented languages. University of Washington Seattle, Technical report 96-06 2 (1996)
7. Chen, S., Erwig, M.: Principal type inference for GADTs. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp. 416–428. ACM, New York (2016)
8. Cheney, J., Hinze, R.: First-class phantom types. Technical report. Cornell University (2003)
9. Chlipala, A.: Certified Programming with Dependent Types. MIT Press, Cambridge (2017). http://adam.chlipala.net/cpdt/
10. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: MultiJava: modular open classes and symmetric multiple dispatch for Java. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 130–145. ACM (2000)
11. Cook, W.R.: Object-oriented programming versus abstract data types. In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (eds.) REX 1990. LNCS, vol. 489, pp. 151–178. Springer, Heidelberg (1991). https://doi.org/10.1007/BFb0019443
12. Cook, W.R.: On understanding data abstraction, revisited. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 557–572. ACM (2009)
13. Coquand, T., Huet, G.: The calculus of constructions. Inf. Comput. **76**(2–3), 95–120 (1988)
14. Curien, P.L., Herbelin, H.: The duality of computation. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000, pp. 233–243. ACM, New York (2000)
15. Danvy, O., Johannsen, J., Zerny, I.: A walk in the semantic park. In: Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, pp. 1–12. ACM, New York (2011)
16. Danvy, O., Millikin, K.: Refunctionalization at work. Sci. Comput. Program. **74**(8), 534–549 (2009)
17. Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: Proceedings of the Conference on Principles and Practice of Declarative Programming, pp. 162–174 (2001)
18. Diehl, S., Hartel, P., Sestoft, P.: Abstract machines for programming language implementation. Future Gener. Comput. Syst. **16**(7), 739–751 (2000)
19. Downen, P., Ariola, Z.M.: The duality of construction. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 249–269. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_14
20. Emir, B., Kennedy, A., Russo, C., Yu, D.: Variance and generalized constraints for C$^\sharp$ generics. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 279–303. Springer, Heidelberg (2006). https://doi.org/10.1007/11785477_18
21. Ernst, M.D., Kaplan, C., Chambers, C.: Predicate dispatching: a unified theory of dispatch. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 186–211. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054092
22. Erwig, M., Jones, S.P.: Pattern guards and transformational patterns. Electron. Notes Theor. Comput. Sci. **41**(1), 3 (2001)
23. Hagino, T.: Codatatypes in ML. J. Symb. Comput. **8**(6), 629–650 (1989)
24. Jacobs, B.: Objects and classes, coalgebraically. In: Freitag, B., Jones, C.B., Lengauer, C., Schek, H.J. (eds.) Object Orientation with Parallelism and Persistence, vol. 370, pp. 83–103. Springer, Boston (1995). https://doi.org/10.1007/978-1-4613-1437-0_5

25. Johnsson, T.: Lambda lifting: transforming programs to recursive equations. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 190–203. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-15975-4_37
26. Kennedy, A., Russo, C.V.: Generalized algebraic data types and object-oriented programming. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 21–40. ACM (2005)
27. Levy, P.B.: Call-by-push-value: a subsuming paradigm. In: Girard, J.-Y. (ed.) TLCA 1999. LNCS, vol. 1581, pp. 228–243. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48959-2_17
28. Oliveira, B.C., Moors, A., Odersky, M.: Type classes as objects and implicits. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2010, pp. 341–360. ACM, New York (2010)
29. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, pp. 50–61. ACM, New York (2006)
30. Pierce, B.C.: Types and Programming Languages. Massachusetts Institute of Technology, Cambridge (2002)
31. Rendel, T., Trieflinger, J., Ostermann, K.: Automatic refunctionalization to a language with copattern matching: with applications to the expression problem. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, pp. 269–279. ACM, New York (2015)
32. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the ACM Annual Conference, pp. 717–740. ACM (1972)
33. Reynolds, J.C.: User-defined types and procedural data structures as complementary approaches to data abstraction. In: Schuman, S. (ed.) New Directions in Algorithmic Languages 1975, pp. 157–168. IFIP Working Group 2.1 on Algol, INRIA, Rocquencourt, France (1975)
34. Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP 2009, pp. 341–352. ACM, New York (2009)
35. Setzer, A., Abel, A., Pientka, B., Thibodeau, D.: Unnesting of copatterns. In: Dowek, G. (ed.) RTA 2014. LNCS, vol. 8560, pp. 31–45. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_3
36. Thibodeau, D., Cave, A., Pientka, B.: Indexed codata types. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, pp. 351–363. ACM, New York (2016)
37. Wadler, P.: The expression problem. Note to Java Genericity mailing list, November 1998
38. Wadler, P.: Call-by-value is dual to call-by-name. In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, pp. 189–201. ACM, New York (2003)
39. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. **115**(1), 38–94 (1994)
40. Xi, H.X., Chiyan, C., Chen, G.: Guarded recursive datatype constructors. In: Proceedings of the Symposium on Principles of Programming Languages, pp. 224–235. ACM (2003)

Deterministic Concurrency: A Clock-Synchronised Shared Memory Approach

Joaquín Aguado[1(✉)], Michael Mendler[1], Marc Pouzet[2], Partha Roop[3], and Reinhard von Hanxleden[4]

[1] Otto-Friedrich-Universität Bamberg, Bamberg, Germany
joaquin.aguado@uni-bamberg.de
[2] École Normale Supérieure, Paris, France
[3] University of Auckland, Auckland, New Zealand
[4] Christian-Albrechts-Universität zu Kiel, Kiel, Germany

Abstract. Synchronous Programming (SP) is a universal computational principle that provides deterministic concurrency. The same input sequence with the same timing always results in the same externally observable output sequence, even if the internal behaviour generates uncertainty in the scheduling of concurrent memory accesses. Consequently, SP languages have always been strongly founded on mathematical semantics that support formal program analysis. So far, however, communication has been constrained to a set of primitive clock-synchronised shared memory (CSM) data types, such as data-flow registers, streams and signals with restricted read and write accesses that limit modularity and behavioural abstractions.

This paper proposes an extension to the SP theory which retains the advantages of deterministic concurrency, but allows communication to occur at higher levels of abstraction than currently supported by SP data types. Our approach is as follows. To avoid data races, each CSM type publishes a *policy interface* for specifying the admissibility and precedence of its access methods. Each instance of the CSM type has to be policy-coherent, meaning it must behave deterministically under its own policy—a natural requirement if the goal is to build deterministic systems that use these types. In a policy-constructive system, all access methods can be scheduled in a policy-conformant way for all the types without deadlocking. In this paper, we show that a policy-constructive program exhibits deterministic concurrency in the sense that all policy-conformant interleavings produce the same input-output behaviour. Policies are conservative and support the CSM types existing in current SP languages. Technically, we introduce a kernel SP language that uses arbitrary policy-driven CSM types. A big-step fixed-point semantics for this language is developed for which we prove determinism and termination of constructive programs.

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 86–113, 2018.
https://doi.org/10.1007/978-3-319-89884-1_4

Keywords: Synchronous programming · Data abstraction
Clock-synchronised shared memory · Determinacy · Concurrency
Constructive semantics

1 Introduction

Concurrent programming is challenging. Arbitrary interleavings of concurrent threads lead to non-determinism with data races imposing significant integrity and consistency issues [1]. Moreover, in many application domains such as safety-critical systems, *determinism* is indeed a matter of life and death. In a medical-device software, for instance, the same input sequence from the sensors (with the same timing) must always result in the same output sequence for the actuators, even if the run-time software architecture regime is unpredictable.

Synchronous programming (SP) delivers *deterministic concurrency* out of the box[1] which explains its success in the design, implementation and validation of embedded, reactive and safety-critical systems for avionics, automotive, energy and nuclear industries. Right now SP-generated code is flying on the Airbus 380 in systems like flight control, cockpit display, flight warning, and anti-icing just to mention a few. The SP mathematical theory has been fundamental for implementing correct-by-construction program-derivation algorithms and establishing formal analysis, verification and testing techniques [2]. For SCADE[2], the SP industrial modelling language and software development toolkit, the formal SP background has been a key aspect for its certification at the highest level A of the aerospace standard DO-178B/C. This SP rigour has also been important for obtaining certifications in railway and transportation (EN 50128), industry and energy (IEC 61508), automotive (TÜV and ISO 26262) as well as for ensuring full compliance with the safety standards of nuclear instrumentation and control (IEC 60880) and medical systems (IEC 62304) [3].

Synchronous Programming in a Nutshell. At the top level, we can imagine an SP system as a black-box with inputs and outputs for interacting with its environment. There is a special input, called the *clock*, that determines when the communication between system and environment can occur. The clock gets an input stimulus from the environment at discrete times. At those moments we say that the clock *ticks*. When there is no tick, there is no possible communication, as if system and environment were disconnected. At every tick, the system *reacts* by reading the current inputs and executing a *step function* that delivers outputs and changes the internal memory. For its part, the environment must synchronise with this reaction and do not go ahead with more ticks. Thus,

[1] Milner's distinction between *determinacy* and *determinism* is that a computation is *determinate* if the same input sequence produces the same output sequence, as opposed to *deterministic* computations which in addition have identical internal behaviour/scheduling. In this paper we use both terms synonymously to mean determinacy in Milner's sense, i.e., observable determinism.
[2] SCADE is a product of ANSYS Inc. (http://www.esterel-technologies.com/).

in SP, we assume (*Synchrony Hypothesis*) that the time interval of a system reaction, also called *macro-step* or (*synchronous*) *instant*, appears instantaneous (has zero-delay) to the environment. Since each system reaction takes exactly one clock tick, we describe the evolution of the system-environment interaction as a synchronous (lock-step) sequence of macro-steps. The SP theory guarantees that all externally observable interaction sequences derived from the macro-step reactions define a functional input-output relation.

The fact that the sequences of macro-steps take place in time and space (memory) has motivated two orthogonal developments of SP. The *data-flow* view regards input-output sequences as synchronous streams of data changing over time and studies the functional relationships between streams. Dually, the *control-flow* approach projects the information of the input-output sequences at each point in time and studies the changes of this global state as time progresses, i.e., from one tick to the next. The SP paradigm includes languages such as Esterel [4], Quartz [5] and SC [6] in the imperative control-flow style and languages like Signal [7], Lustre [8] and Lucid Synchrone [9] that support the declarative data-flow view. There are even mixed control-data flow language such as Esterel V7 [10] or SCADE [3]. Independently of the execution model, the common strength to all of these SP languages is a precise formal semantics—an indispensable feature when dealing with the complexities of concurrency.

At a more concrete level, we can visualise an SP system as a white-box where inside we find (graphical or textual) code. In the SP domain, the program must be divided into fragments corresponding to the macro-step reactions that will be executed instantaneously at each tick. Declarative languages usually organise these macro-steps by means of (internally generated) activation clocks that prescribe the blocks (nodes) that are performed at each tick. Instead, imperative textual languages provide a `pause` statement for explicitly delimiting code execution within a synchronous instant. In either case, the Synchrony Hypothesis conveniently abstracts away all the, typically concurrent, low-level *micro-steps* needed to produce a system reaction. The SP theory explains how the micro-step accesses to shared memory must be controlled so as to ensure that all internal (white-box) behaviour eventually stabilises, completing a deterministic macro-step (black-box) response. For more details on SP, the reader is referred to [2].

State of the Art. Traditional imperative SP languages provide constructs to model control-dominated systems. Typically, these include a concurrent composition of *threads* (sequential processes) that guarantees determinism and offers *signals* as the main means for data communication between threads. Signals behave like shared variables for which the concurrent accesses occurring within a macro-step are scheduled according to the following principles: A *pure signal* has a *status* that can be *present* (1) or *absent* (0). At the beginning of each macro-step, pure signals have status 0 by default. In any instant, a signal `s` can be explicitly *emitted* with the statement `s.emit`() which atomically sets its status to 1. We can read the status of `s` with the statement `s.pres`(), so the control-flow can branch depending on run-time signal statuses. Specifically, inside programs, if-then-else constructions await for the appropriate combination of present and absent signal statuses to emit (or not) more signals. The main issue is to avoid inconsistencies due to circular causality resulting from decisions based on absent statuses. Thus, the order in which the access methods `emit`, `pres` are scheduled matters for the final result. The usual SP rule for ensuring determinism is that the `pres` test must wait until the final signal status is decided. If all signal accesses can be scheduled in this *decide-then-read* way then the program is *constructive*. All schedules that keep the decide-then-read order will produce the same input-output result. This is how SP reconciles concurrency and observable determinism and generates much of its algebraic appeal. Constructiveness of programs is what static techniques like the *must-can* analysis [4, 11–13] verify although in a more abstract manner. Pure signals are a simple form of *clock-synchronised shared memory* (CSM) data types with access methods (operations) specific to this CSM type. Existing SP control-flow languages also support other restricted CSM types such valued signals and arrays [10] or sequentially constructive variables [6].

Contribution. This paper proposes an extension to the SP model which retains the advantages of deterministic concurrency while widening the notion of constructiveness to cover more general CSM types. This allows shared-memory communication to occur at higher levels of abstraction than currently supported. In particular, our approach subsumes both the notions of *Berry-constructiveness* [4] for Esterel and *sequential constructiveness* for SCL [14]. This is the first time that these SP communication principles are combined side-by-side in a single language. Moreover, our theory permits other predefined communication structures to coexist safely under the same uniform framework, such as data-flow variables [8], registers [15], Kahn channels [16], priority queues, arrays as well as other CSM types currently unsupported in SP.

Synopsis and Overview. The core of our approach is presented in Sect. 2 where *policies* are introduced as a (constructive) synchronisation mechanism for arbitrary *abstract data types* (ADT). For instance, the policy of a pure signal is depicted in Fig. 1. It has two control *states* 0 and 1 corresponding to the two possible signal statuses. Transitions are decorated with method names `pres`, `emit` or with σ to indicate a clock tick.

The policy tells us whether a given method or tick is *admissible*, i.e., if it can be scheduled from a particular state[3]. In addition, transitions include a *blocking* set of method names as part of their *action* labels. This set determines a *precedence* between methods from a given state. A label $m : L$ specifies that all methods in L take precedence over m. An empty blocking set $\emptyset$ indicates no precedences. To improve visualisation, we

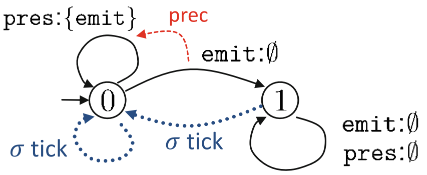


Fig. 1. Pure signal policy.

[3] The signal policy in Fig. 1 does not impose any admissibility restriction since methods `pres` and `emit` can be scheduled from every policy state.

highlight precedences by dotted (red) arrows tagged `prec`[4]. The *policy interface* in Fig. 1 specifies the decide-then-read protocol of pure signals as follows. At any instant, if the signal status is 0 then the `pres` test can only be scheduled if there are no more potential `emit` statements that can still update the status to 1. This explains the precedence of the `emit` transition over the self loop with action label `pres` : {`emit`} from state 0. Conversely, transitions `pres` and `emit` from state 1 have no precedences, meaning that the `pres` and `emit` methods are *confluent* so they can be freely scheduled (interleaved). The reason is that a signal status 1 is already decided and can no longer be changed by either method in the same instant. In general, any two admissible methods that do not block each other must be confluent in the sense that the same policy state is reached independently of their order of execution. Note that all the σ transition go to the *initial* state 0 since at each tick the SP system enters a new macro-step where all pure signals get initialised to the 0 status.

Section 2 describes in detail the idea of a scheduling policy on general CSM types. This leads to a type-level *coherence* property, which is a local form of determinism. Specifically, a CSM type is *policy-coherent* if it satisfies the (policy) specification of admissibility and precedence of its access methods. The point is that a policy-coherent CSM type per se behaves deterministically under its own policy—a very natural requirement if the goal is to build deterministic systems that use this type. For instance, the fact that Esterel signals are deterministic (policy-coherent) in the first place permits techniques such as the must-can analysis to get constructive information about deterministic programs. We show how policy-coherence implies a global determinacy property called *commutation*. Now, in a *policy-constructive* program all access methods can be scheduled in a *policy-conforming* way for all the CSM types without deadlocking. We also show that, for policy-coherent types, a policy-constructive program exhibits deterministic concurrency in the sense that all policy-conforming interleavings produce the same input-output behaviour.

To implement a constructive scheduling mechanism parameterised in arbitrary CSM type policies, we present the synchronous kernel language, called *Deterministic Concurrent Language* (DCoL), in Sect. 2.1. DCoL is both a minimal language to study the new mathematical concepts but can also act as an intermediate language for compiling existing SP Sect. 3 presents its policy-driven operational semantics for which determinacy and termination is proven. Section 3 also explains how this model generalises existing notions of constructiveness. We discuss related work in Sect. 4 and present our conclusions in Sect. 5.

A companion of this paper is the research report (https://www.uni-bamberg.de/fileadmin/uni/fakultaeten/wiai_professuren/grundlagen_informatik/papersMM/report-WIAI-102-Feb-2018.pdf) [17] which contains detailed proofs and additional examples of CSM types.

[4] We tacitly assume that the tick transitions σ have the lowest *priority* since only when the reaction is over, the clock may tick. We could be more explicit and write σ : {`pres`, `emit`} as action labels for these transitions.

2 Synchronous Policies

This section introduces a kernel synchronous *Deterministic Concurrent Language* (DCoL) for policy-conformant constructive scheduling which integrates policy-controlled CSM types within a simple syntax. DCoL is used to discuss the behavioural (clock) abstraction limitations of current SP. Then policies are introduced as a mechanism for specifying the scheduling discipline for CSM types which, in this form, can encapsulate arbitrary ADTs.

2.1 Syntax

The syntax of DCoL is given by the following operators:

$P ::= \mathtt{skip}$	instantaneous termination
$\mid\ \mathtt{pause}$	wait for next instant (clock tick)
$\mid\ P \parallel P$	parallel composition
$\mid\ P\ ;\ P$	sequential composition
$\mid\ \mathtt{let}\, x = \mathtt{c}.m(e)\, \mathtt{in}\, P$	access method call, x value variable
$\mid\ \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P$	conditional branching, e value expression
$\mid\ \mathsf{rec}\, p.\, P$	recursive closure
$\mid\ p$	process variable

The first two statements correspond to the two forms of immediate *completion*: `skip` terminates instantaneously and `pause` waits for the logical clock to terminate. The operators $P \parallel Q$ and $P\ ;\ Q$ are *parallel interleaving* and *imperative sequential* composition of threads with the standard operational interpretation. Reading and destructive updating is performed through the execution of method calls $\mathtt{c}.m(e)$ on a CSM *variable* $\mathtt{c} \in \mathsf{O}$ with a method $m \in \mathsf{M}_\mathtt{c}$. The sets O and $\mathsf{M}_\mathtt{c}$ define the granularity of the available memory accesses. The construct $\mathtt{let}\, x = \mathtt{c}.m(e)\, \mathtt{in}\, P$ calls m on $\mathtt{c}$ with an input parameter determined by *value expression* e. It binds the return value to variable x and then executes program P, which may depend on x, sequentially afterwards. The execution of $\mathtt{c}.m(e)$ in general has the side-effect of changing the internal memory of $\mathtt{c}$. In contrast, the evaluation of expression e is side-effect free. For convenience we write $x = \mathtt{c}.m(e)\,;P$ for $\mathtt{let}\, x = \mathtt{c}.m(e)\, \mathtt{in}\, P$. When P does not depend on x then we write $\mathtt{c}.m(e)\,;P$ and $\mathtt{c}.m(e)\,;$ for $\mathtt{c}.m(e)\,;\,\mathtt{skip}$. The exact syntax of value expressions e is irrelevant for this work and left open. It could be as simple as permitting only constant value literals or a full-fledged functional language. The *conditional* $\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P$ has the usual interpretation. For simplicity, we may write $\mathtt{if}\ \mathtt{c}.m(e)\ \mathtt{then}\ P\ \mathtt{else}\ Q$ to mean $x = \mathtt{c}.m(e)\,;\ \mathtt{if}\ x\ \mathtt{then}\ P\ \mathtt{else}\ Q$. The *recursive closure* $\mathsf{rec}\, p.\, P$ binds the behaviour P to the program label p so it can be called from within P. Using this construct we can build iterative behaviours. For instance, $\mathtt{loop}\, P\, \mathtt{end}\ =_{df}$ $\mathsf{rec}\, p.\, P\,;\ \mathtt{pause}\ ;p$ indefinitely repeats P in each tick. We assume that in a closure $\mathsf{rec}\, p.\, P$ the label p is (i) *clock guarded*, i.e., it occurs in the scope of at least one `pause` (meaning no instantaneous loops) and (ii) all occurrences of p are in the same thread. Thus, $\mathsf{rec}\, p.\, p$ is illegal because of (i) and $\mathsf{rec}\, p.\, (\mathtt{pause}\ ;\ p \parallel \mathtt{pause}\ ;\ p)$ is not permitted because of (ii).

This syntax seems minimalistic compared to existing SP languages. For instance, it does not provide primitives for pre-emption, suspension or traps as in Quartz or Esterel. Recent work [18] has shown how these control primitives can be translated into the constructs of the SCL language, exploiting destructive update of sequentially constructive (SC) variables. Since SC variables are a special case of policy-controlled CSM variables, DCoL is at least as expressive as SCL.

2.2 Limited Abstraction in SP

The pertinent feature of standard SP languages is that they do not permit the programmer to express sequential execution order inside a tick, for destructive updates of signals. All such updates are considered concurrent and thus must either be combined or concern distinct signals. For instance, in languages such as Esterel V7 or Quartz, a parallel composition

$$(v = \mathtt{xs.read}() \,;\, \mathtt{ys.emit}(v+1)) \;\|\; (\mathtt{xs.emit}(1) \,;\, \mathtt{xs.emit}(5)) \qquad (1)$$

of signal emissions is only constructive if a commutative and associative function is defined on the shared signal `xs` to combine the values assigned to it. But then, by the properties of this *combination function*, we get the same behaviour if we swap the assignments of values 1 and 5, or execute all in parallel as in

$$v = \mathtt{xs.read}() \;\|\; \mathtt{ys.emit}(v+1) \;\|\; \mathtt{xs.emit}(1) \;\|\; \mathtt{xs.emit}(5).$$

If what we intended with the second emission $\mathtt{xs.emit}(5)$ in (1) was to override the first $\mathtt{xs.emit}(1)$ like in normal imperative programming so that the concurrent thread $v = \mathtt{xs.read}() \,;\, \mathtt{ys.emit}(v+1)$ will read the updated value as $v = 5$? Then we need to introduce a `pause` statement to separate the emissions by a clock tick and delay the assignment to `ys` as in

$$(\mathtt{pause} \,;\, v = \mathtt{xs.read}() \,;\, \mathtt{ys.emit}(v+1)) \;\|\; (\mathtt{xs.emit}(1) \,;\, \mathtt{pause} \,;\, \mathtt{xs.emit}(5)).$$

This makes behavioural abstraction difficult. For instance, suppose `nats` is a synchronous reaction module, possibly composite and with its own internal clocking, which returns the stream of natural numbers. Every time its step function $\mathtt{nats.step}()$ is called it returns the next number and increments its internal state. If we want to pair up two successive numbers within one tick of an outer clock and emit them in a single signal `ys` we would write something like $x_1 = \mathtt{nats.step}() \,;\, x_2 = \mathtt{nats.step}() \,;\, \mathtt{y.emit}(x_1, x_2)$ where x_1, x_2 are thread-local value variables. This over-clocking is impossible in traditional SP because there is no imperative sequential composition by virtue of which we can call the step function of the same module instance twice within a tick. Instead, the two calls $\mathtt{nats.step}()$ are considered concurrent and thus create non-determinacy in the value of `y`.[5] To avoid a compiler error we must separate the calls by a clock as

[5] In Esterel V7 it is possible to use a module twice in a "sequential" composition $x_1 = \mathtt{nats.step}(); x_2 = \mathtt{nats.step}()$. However, the two occurrences of `nats` are distinct instances with their own internal state. Both calls will thus return the same value.

in x_1 = `nats.step()` ; `pause` ; x_2 = `nats.step()` ; `y. emit`(x_1, x_2) which breaks the intended clock abstraction.

The data abstraction limitation of traditional SP is that it is not directly possible to encapsulate a composite behaviour on synchronised signals as a shared synchronised object. For this, the simple decide-then-read signal protocol must be generalised, in particular, to distinguish between concurrent and sequential accesses to the shared data structure. A concurrent access x_1 = `nats.step()` ∥ x_2 = `nats.step()` must give the same value for x_1 and x_2, while a sequential access x_1 = `nats.step()` ; x_2 = `nats.step()` must yield successive values of the stream. In a sequence x = `xs.read()` ; `xs. emit`(v) the x does not see the value v but in a parallel x = `xs.read()` ∥ `xs. emit`(v) we may want the read to wait for the emission. The rest of this section covers our theory on policies in which this is possible. The modularity issue is reconsidered in Sect. 2.6.

2.3 Concurrent Access Policies

In the white-box view of SP, an imperative program consists of a set of threads (sequential processes) and some CSM variables for communication. Due to concurrency, a given *thread under control* (TUC) has the chance to access the shared variables only from time to time. For a given CSM variable, a *concurrent access policy* (CAP) is the locking mechanism used to control the accesses of the current TUC and its environment. The locking is to ensure that determinacy of the CSM type is not broken by the concurrent accesses. A CAP is like a policy which has extra transitions to model potential environment accesses outside the TUC. Concretely, a CAP is given by a state machine where each transition label $a : L$ codifies an *action* a taking place on the shared variable with *blocking set* L, where L is a set of methods that take precedence over a. The action is either a *method* $m : L$, a *silent action* $\tau : L$ or a *clock tick* $\sigma : L$. A transition $m : L$ expresses that in the current CAP control state, the method m can be called by the TUC, provided that no method in L is called concurrently. There is a *Determinacy Requirement* that guarantees that each method call by the TUC has a blocking set and successor state. Additionally, the execution of methods by the CAP must be *confluent* in the sense that if two methods are admissible and do not block each other, then the CAP reaches the same policy state no matter the order in which they are executed. This is to preserve determinism for concurrent variable accesses. A transition $\tau : L$ internalises method calls by the TUC's concurrent environment which are uncontrollable for the TUC. In the sequel, the actions in $\mathsf{M_c} \cup \{\sigma\}$ will be called *observable.* A transition $\sigma : L$ models a clock synchronisation step of the TUC. Like method calls, such clock ticks must be determinate as stated by the Determinacy Requirement. Additionally, the clock must always wait for any predicted concurrent τ-activity to complete. This is the *Maximal Progress Requirement.* Note that we do not need confluence for clock transitions since they are not concurrent.

Definition 1. *A* concurrent access policy *(*CAP*)* $\Vdash_{\mathsf{c}}$ *of a* CSM *variable* c *with (access) methods* $\mathsf{M_c}$ *is a state machine consisting of a set of* control states $\mathbb{P}_{\mathsf{c}}$, *an* initial state $\varepsilon \in \mathbb{P}_c$ *and a labelled* transition relation $\rightarrow \subseteq \mathbb{P}_c \times A_c \times \mathbb{P}_c$ *with* action *labels* $A_c = (M_c \cup \{\tau, \sigma\}) \times 2^{M_c}$. *Instead of* $(\mu_1, (a, L), \mu_2) \in \rightarrow$ *we write* $\mu_1 \xrightarrow{a:L} \mu_2$. *We then say action* a *is* admissible *in state* μ_1 *and* blocked *by all methods* $m \in L \subseteq M_c$. *When the* blocking set L *is irrelevant we drop it and write* $\mu_1 \xrightarrow{a} \mu_2$. *A* CAP *must satisfy the following conditions:*

– Determinacy. *If* $\mu \xrightarrow{a:L_1} \mu_1$ *and* $\mu \xrightarrow{a:L_2} \mu_2$ *then* $L_1 = L_2$ *and* $\mu_1 = \mu_2$ *provided* a *is observable, i.e.,* $a \neq \tau$.
– Confluence. *If* $\mu \xrightarrow{m_1:L_1} \mu_1$ *and* $\mu \xrightarrow{m_2:L_2} \mu_2$ *do not block each other, i.e.,* $m_1 \in M_c \setminus L_2$ *and* $m_2 \in M_c \setminus L_1$, *then for some* μ' *both* $\mu_1 \xrightarrow{m_2} \mu'$ *and* $\mu_2 \xrightarrow{m_1} \mu'$.
– Maximal Progress. $\mu \xrightarrow{a:L_1} \mu_1$ *and* $\mu \xrightarrow{\sigma:L_2} \mu_2$ *imply* a *is observable.*

A policy *is a* CAP *without any (concurrent)* τ *activity, i.e., every* $\mu \xrightarrow{a} \mu'$ *implies that* a *is observable.* □

The use of a CAP as a concurrent policy arises from the notion of enabling. Informally, an observable action $a \in \mathsf{M}_c \cup \{\sigma\}$ is enabled in a state μ of a CAP if it is admissible in μ *and* in all subsequent states reachable under arbitrary silent steps. To formalise this we define *weak transitions* $\mu_1 \Rightarrow \mu_2$ inductively to express that either $\mu_1 = \mu_2$ or $\mu_1 \Rightarrow \mu'$ and $\mu' \xrightarrow{\tau} \mu_2$. We exploit the determinacy for observable actions $a \in \mathsf{M}_c \cup \{\sigma\}$ and write $\mu \odot a$ for the unique μ' such that $\mu \xrightarrow{a} \mu'$, if it exists.

Definition 2. *Given a* CAP $\Vdash_c = (\mathbb{P}_c, \varepsilon, \longrightarrow)$, *an observable action* $a \in M_c \cup \{\sigma\}$ *is* enabled *in state* $\mu \in \mathbb{P}_c$, *written* $\mu \Vdash_c \downarrow a$, *if* $\mu' \odot a$ *exists for all* μ' *such that* $\mu \Rightarrow \mu'$. *A sequence* $\boldsymbol{a} \in (M_c \cup \{\sigma\})^*$ *of observable actions is* enabled *in* $\mu \in \mathbb{P}_c$, *written* $\mu \Vdash_c \downarrow \boldsymbol{a}$, *if (i)* $\boldsymbol{a} = \varepsilon$ *or (ii)* $\boldsymbol{a} = a\,\boldsymbol{b}$, $\mu \Vdash_c \downarrow a$ *and* $\mu \odot a \Vdash_c \downarrow \boldsymbol{b}$. □

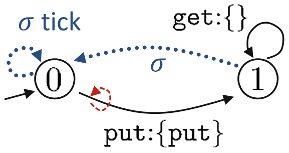


Fig. 2. Synchronous IVar.

Example 1. Consider the policy $\Vdash_\mathtt{s}$ in Fig. 1 of an Esterel pure signal $\mathtt{s}$. An edge labelled $a{:}L$ from state μ_1 to μ_2 corresponds to a transition $\mu_1 \xrightarrow{a:L} \mu_2$ in $\Vdash_\mathtt{s}$. The start state is $\varepsilon = 0$ and the methods $\mathsf{M}_\mathtt{s} = \{\mathtt{pres}, \mathtt{emit}\}$ are always admissible, i.e., $\mu \odot m$ is defined in each state μ for all methods m. The presence test does not change the state and any emission sets it to 1, i.e., $\mu \odot \mathtt{pres} = \mu$ and $\mu \odot \mathtt{emit} = 1$ for all $\mu \in \mathbb{P}_\mathtt{s}$. Each signal status is reset to 0 with the clock tick, i.e., $\mu \odot \sigma = 0$. Clearly, $\Vdash_\mathtt{s}$ satisfies Determinacy. A presence test on a signal that is not emitted yet has to wait for all pending concurrent emissions, that is $\mathtt{emit}$ blocks $\mathtt{pres}$ in state 0, i.e., $0 \xrightarrow{\mathtt{pres}:\{\mathtt{emit}\}} 0$. Otherwise, no transition is blocked. Also, all competing transitions $\mu \xrightarrow{m_1:L_1} \mu_1$ and $\mu \xrightarrow{m_2:L_2} \mu_2$ that do not block each other, are of the form $\mu_1 = \mu_2$, from which Confluence follows. Note that since there are no silent transitions, Maximal Progress is always fulfilled too. Moreover, an action sequence is enabled in a state μ (Definition 2) iff it corresponds to a path in the automaton starting from μ. Hence, for $\boldsymbol{m} \in \mathsf{M}_\mathtt{s}^*$ we have

$0 \Vdash_{\mathsf{s}} \downarrow \boldsymbol{m}$ iff $\boldsymbol{m}$ is in the regular language[6] $\mathtt{pres}^* + \mathtt{pres}^*\,\mathtt{emit}(\mathtt{pres} + \mathtt{emit})^*$ and $1 \Vdash_{\mathsf{s}} \downarrow \boldsymbol{m}$ for all $\boldsymbol{m} \in \mathsf{M}_{\mathsf{s}}^*$.

Contrast $\Vdash_{\mathsf{s}}$ with the policy $\Vdash_{\mathsf{c}}$ of a synchronous *immutable variable* (IVar) c shown in Fig. 2 with methods $\mathsf{M}_{\mathsf{c}} = \{\mathtt{get}, \mathtt{put}\}$. During each instant an IVar can be written (put) at most once and cannot be read (get) until it has been written. No value is stored between ticks, which means the memory is only temporary and can be reused, e. g., IVars can be implemented by wires. Formally, $\mu \Vdash_{\mathsf{c}} \downarrow \mathtt{put}$ iff $\mu = 0$, where 0 is the initial empty state and $\mu \Vdash_{\mathsf{c}} \downarrow \mathtt{get}$ iff $\mu = 1$, where 1 is the filled state. The transition $0 \xrightarrow{\mathtt{put}:\{\mathtt{put}\}} 1$ switches to filled state where get is admissible but put is not, anymore. The blocking $\{\mathtt{put}\}$ means there cannot be other concurrent threads writing c at the same time. □

2.4 Enabling and Policy Conformance

A policy describes what a single thread can do to a CSM variable c when it operates alone. In a CAP all potential activities of the environment are added as τ-transitions to block the TUC's accesses. To implement this τ-locking we define an operation that generates a CAP $[\mu, \gamma]$ out of a policy. In this construction, $\mu \in \mathbb{P}_{\mathsf{c}}$ is a policy state recording the history of methods that have been performed on c so far (*must* information). The second component $\gamma \subseteq \mathsf{M}_{\mathsf{c}}^*$ is a prediction for the sequences of methods that can still potentially be executed by the concurrent environment (*can* information).

Definition 3. *Let $(\mathbb{P}_{\mathsf{c}}, \varepsilon, \rightarrow)$ be a policy. We define a* CAP $\Vdash_{\mathsf{c}}$ *where states are pairs $[\mu, \gamma]$ such that $\mu \in \mathbb{P}_{\mathsf{c}}$ is a policy state and $\gamma \subseteq M_{\mathsf{c}}^*$ is a prediction. The initial state is $[\varepsilon, M_{\mathsf{c}}^*]$ and the transitions are as follows:*

1. *The observable transitions $[\mu_1, \gamma_1] \xrightarrow{m:L} [\mu_2, \gamma_2]$ are such that $\gamma_1 = \gamma_2$ and $\mu_1 \xrightarrow{m:L} \mu_2$ provided that for all sequences $n\,\boldsymbol{n} \in \gamma_1$ with $\mu_1 \xrightarrow{n} \mu'$ we have $n \notin L$.*
2. *The silent transitions are $[\mu_1, \gamma_1] \xrightarrow{\tau:L} [\mu_2, \gamma_2]$ such that $\emptyset \neq m\,\gamma_2 \subseteq \gamma_1$ and $\mu_1 \xrightarrow{m:L} \mu_2$.*
3. *The clock transitions are $[\mu_1, \gamma_1] \xrightarrow{\sigma:L} [\mu_2, \gamma_2]$ such that $\gamma_1 = \emptyset$ and $\mu_1 \xrightarrow{\sigma:L} \mu_2$.* □

Silent steps arise from the concurrent environment: A step $[\mu_1, \gamma_1] \xrightarrow{\tau:L} [\mu_2, \gamma_2]$ removes some prefix method m from the environment prediction γ_1, which contracts to an updated suffix prediction γ_2 with $m\,\gamma_2 \subseteq \gamma_1$. This method m is executed on the CSM variable, changing the policy state to $\mu_2 = \mu_1 \odot m$. A method m is enabled, $[\mu, \gamma] \Vdash_{\mathsf{c}} \downarrow m$, if for all $[\mu_1, \gamma_1]$ which are τ-reachable from $[\mu, \gamma]$, method m is admissible, i.e., $[\mu_1, \gamma_1] \xrightarrow{m} [\mu_2, \gamma_1]$ for some μ_2.

Example 2. Consider concurrent threads $P_1 \parallel P_2$, where $P_2 = \mathtt{zs.put}(5)\,;\, u = \mathtt{ys.get}()$ and $P_1 = v = \mathtt{zs.get}()\,;\, \mathtt{ys.put}(v + 1)$ with IVars zs, ys according to

[6] We are more liberal than Esterel where emit cannot be called sequentially after pres.

Example 1. Under the IVar policy the execution is deterministic, so that first P_2 writes on $\mathtt{zs}$, then P_1 reads from $\mathtt{zs}$ and writes to $\mathtt{ys}$, whereupon finally P_1 reads $\mathtt{ys}$. Suppose the variables have reached policy states $\mu_{\mathtt{zs}}$ and $\mu_{\mathtt{ys}}$ and the threads are ready to execute the residual programs P'_i waiting at some method call $\mathtt{c}_i.m_i(v_i)$, respectively. Since thread P'_i is concurrent with the other P'_{3-i}, it can only proceed if m_i is not blocked by P'_{3-i}, i.e., if $[\mu_{\mathtt{c}_i}, can_{\mathtt{c}_i}(P'_{3-i})] \Vdash_{\mathtt{c}_i} \downarrow m_i$, where $can_{\mathtt{c}}(P) \subseteq \mathsf{M}^*_{\mathtt{c}}$ is the set of method sequences predicted for P on $\mathtt{c}$.

Initially we have $\mu_{\mathtt{zs}} = 0 = \mu_{\mathtt{ys}}$. Since method $\mathtt{get}$ is not admissible in state 0, we get $[0, can_{\mathtt{zs}}(P_2)] \nVdash_{\mathtt{zs}} \downarrow \mathtt{get}$ by Definitions 3 and 2. So, P_1 is blocked. The $\mathtt{zs.put}$ of P_2, however, can proceed. First, since no predicted method sequence $can_{\mathtt{zs}}(P_1) = \{\mathtt{get}\}$ of P_1 starts with $\mathtt{put}$, the transition $0 \xrightarrow{\mathtt{put}:\{\mathtt{put}\}} 1$ implies that $[0, can_{\mathtt{zs}}(P_1)] \xrightarrow{\mathtt{put}:\{\mathtt{put}\}} [1, can_{\mathtt{zs}}(P_1)]$ by Definition 3(1). Moreover, since $\mathtt{get}$ of P_1 is not admissible in 0, there are no silent transitions out of $[0, can_{\mathtt{zs}}(P_1)]$ according to Definition 3(2). Thus, $[0, can_{\mathtt{zs}}(P_1)] \Vdash_{\mathtt{zs}} \downarrow \mathtt{put}$, as claimed.

When the $\mathtt{zs.put}$ is executed by P_2 it turns into $P'_2 = u \,\mathtt{=}\, \mathtt{ys.get}()$ and the policy state for $\mathtt{zs}$ advances to $\mu'_{\mathtt{zs}} = 1$, while $\mathtt{ys}$ is still at $\mu_{\mathtt{ys}} = 0$. Now $\mathtt{ys.get}$ of P'_2 blocks for the same reason as $\mathtt{zs}$ was blocked in P_1 before. But since P_2 has advanced, its prediction on $\mathtt{zs}$ reduces to $can_{\mathtt{zs}}(P'_2) = \emptyset$. Therefore, the transition $1 \xrightarrow{\mathtt{get}:\emptyset} 1$ implies $[1, can_{\mathtt{zs}}(P'_2)] \xrightarrow{\mathtt{get}:\emptyset} [1, can_{\mathtt{zs}}(P'_2)]$ by Definition 3(1). Also, there are no silent transitions out of $[1, can_{\mathtt{zs}}(P'_2)]$ by Definition 3(2) and so $[\mu'_{\mathtt{zs}}, can_{\mathtt{zs}}(P'_2)] \Vdash_{\mathtt{zs}} \downarrow \mathtt{get}$ by Definition 2. This permits P_1 to execute $\mathtt{zs.get}()$ and proceed to $P'_1 = \mathtt{ys.put}(5+1)$. The policy state of $\mathtt{zs}$ is not changed by this, neither is the state of $\mathtt{ys}$, whence P'_2 is still blocked. Yet, we have $[\mu_{\mathtt{ys}}, can_{\mathtt{zs}}(P'_2)] \Vdash_{\mathtt{ys}} \downarrow \mathtt{put}$ which lets P'_1 complete $\mathtt{ys.put}$. It reaches P''_1 with $can_{\mathtt{ys}}(P''_1) = \emptyset$ and changes the policy state of $\mathtt{ys}$ to $\mu'_{\mathtt{ys}} = 1$. At this point, $[\mu'_{\mathtt{ys}}, can_{\mathtt{zs}}(P''_1)] \Vdash_{\mathtt{ys}} \downarrow \mathtt{get}$ which means P'_2 unblocks to execute $\mathtt{ys.get}$. □

Definition 4. *Let $\Vdash_{\mathtt{c}}$ be a policy for* $\mathtt{c}$. *A method sequence $\boldsymbol{m}_1$* blocks *another $\boldsymbol{m}_2$ in state μ, written $\mu \Vdash_{\mathtt{c}} \boldsymbol{m}_1 \rightarrow \boldsymbol{m}_2$, if $\mu \Vdash_{\mathtt{c}} \downarrow \boldsymbol{m}_2$ but $[\mu, \{\boldsymbol{m}_1\}] \nVdash_{\mathtt{c}} \downarrow \boldsymbol{m}_2$. Two method sequences $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ are* concurrently enabled, *denoted $\mu \Vdash_{\mathtt{c}} \boldsymbol{m}_1 \diamond \boldsymbol{m}_2$ if $\mu \Vdash_{\mathtt{c}} \downarrow \boldsymbol{m}_1$, $\mu \Vdash_{\mathtt{c}} \downarrow \boldsymbol{m}_2$ and both $\mu \nVdash_{\mathtt{c}} \boldsymbol{m}_1 \rightarrow \boldsymbol{m}_2$ and $\mu \nVdash_{\mathtt{c}} \boldsymbol{m}_2 \rightarrow \boldsymbol{m}_1$.* □

Our operational semantics will only let a TUC execute a sequence $\boldsymbol{m}$ provided $[\mu, \gamma] \Vdash_{\mathtt{c}} \downarrow \boldsymbol{m}$, where μ is the current policy state of $\mathtt{c}$ and γ the predicted activity in the TUC's concurrent environment. Symmetrically, the environment will execute any $\boldsymbol{n} \in \gamma$ only if it is enabled with respect to $\boldsymbol{m}$, i.e., if $[\mu, \{\boldsymbol{m}\}] \Vdash \downarrow \boldsymbol{n}$. This means $\mu \Vdash_{\mathtt{c}} \boldsymbol{m} \diamond \boldsymbol{n}$. Policy coherence (Definition 5 below) then implies that every interleaving of the sequences $\boldsymbol{m}$ and any $\boldsymbol{n} \in \gamma$ leads to the same return values and final variable state (Proposition 1).

2.5 Coherence and Determinacy

A *method call* $m(v)$ combines a method $m \in \mathsf{M}_{\mathtt{c}}$ with a method parameter[7] $v \in \mathbb{D}$, where $\mathbb{D}$ is a universal domain for method arguments and return values,

[7] This is without loss of generality since $\mathbb{D}$ may contain value tuples.

including the special don't care value $\_ \in \mathbb{D}$. We denote by $\mathsf{A}_\mathtt{c} = \{m(v) \mid m \in \mathsf{M}_\mathtt{c}, v \in \mathbb{D}\}$ the set of all method calls on object $\mathtt{c}$. Sequences of method calls $\alpha \in \mathsf{A}_\mathtt{c}^*$ can be abstracted back into sequences of methods $\alpha^\# \in \mathsf{M}_\mathtt{c}^*$ by dropping the method parameters: $\varepsilon^\# = \varepsilon$ and $(m(v)\,\alpha)^\# = m\,\alpha^\#$.

Coherence concerns the semantics of method calls as state transformations. Let $\mathbb{S}_\mathtt{c}$ be the domain of memory states of the object $\mathtt{c}$ with initial state $init_\mathtt{c} \in \mathbb{S}_\mathtt{c}$. Each method call $m(v) \in \mathsf{A}_\mathtt{c}$ corresponds to a semantical action $[\![m(v)]\!]_\mathtt{c} \in \mathbb{S}_\mathtt{c} \to (\mathbb{D} \times \mathbb{S}_\mathtt{c})$. If $s \in \mathbb{S}_\mathtt{c}$ is the current state of the object then executing a call $m(v)$ on $\mathtt{c}$ returns a pair $(u, s') = [\![m(v)]\!]_\mathtt{c}(s)$ where the first projection $u \in \mathbb{D}$ is the return value from the call and the second projection $s' \in \mathbb{S}_\mathtt{c}$ is the new updated state of the variable. For convenience, we will denote $u = \pi_1[\![m(v)]\!]_\mathtt{c}(s)$ by $u = s.m(v)$ and $s' = \pi_2[\![m(v)]\!]_\mathtt{c}(s)$ by $s' = s \odot m(v)$. The action notation is extended to sequences of calls $\alpha \in \mathsf{A}_\mathtt{c}^*$ in the natural way: $s \odot \varepsilon = s$ and $s \odot (m(v)\,\alpha) = (s \odot m(v)) \odot \alpha$.

For policy-based scheduling we assume an abstraction function mapping a memory state $s \in \mathbb{S}_\mathtt{c}$ into a policy state $s^\# \in \mathbb{P}_\mathtt{c}$. Specifically, $init_\mathtt{c}^\# = \varepsilon$. Further, we assume the abstraction commutes with method execution in the sense that if we execute a sequence of calls and then abstract the final state, we get the same as if we executed the policy automaton on the abstracted state in the first place. Formally, $(s \odot \alpha)^\# = s^\# \odot \alpha^\#$ for all $s \in \mathbb{S}_\mathtt{c}$ and $\alpha \in A_\mathtt{c}^*$.

Definition 5 (Coherence). *A* CSM *variable* $\mathtt{c}$ *is* policy-coherent *if for all method calls* $a, b \in A_\mathtt{c}$ *whenever* $s^\# \Vdash_\mathtt{c} a^\# \diamond b^\#$ *for a state* $s \in \mathbb{S}_\mathtt{c}$*, then* a *and* b *are* confluent *in the sense that* $s.a = (s \odot b).a$*,* $s.b = (s \odot a).b$ *and* $s \odot a \odot b = s \odot b \odot a$*.* □

Example 3. Esterel pure signals do not carry any data value, so their memory state coincides with the policy state, $\mathbb{S}_\mathtt{s} = \mathbb{P}_\mathtt{s} = \{0, 1\}$ and $s^\# = s$. An emission $\mathtt{emit}$ does not return any value but sets the state of $\mathtt{s}$ to 1, i.e., $s.\mathtt{emit}(\_) = \_ \in \mathbb{D}$ and $s \odot \mathtt{emit}(\_) = 1 \in \mathbb{S}_\mathtt{s}$. A present test returns the state, $s.\mathtt{pres}(\_) = s$, but does not modify it, $s \odot \mathtt{pres}(\_) = s$. From the policy Fig. 1 we find that the concurrent enablings $s^\# \Vdash_\mathtt{s} a^\# \diamond b^\#$ according to Definition 4 are (i) $a = b \in \{\mathtt{pres}(\_), \mathtt{emit}(\_)\}$ for arbitrary s, or (ii) $s = 1$, $a = \mathtt{emit}(\_)$ and $b = \mathtt{pres}(\_)$. In each of these cases we verify $s.a = (s \odot b).a$, $s.b = (s \odot a).b$ and $s \odot a \odot b = s \odot b \odot a$ without difficulty. Note that $1 \Vdash_\mathtt{s} \mathtt{emit} \diamond \mathtt{pres}$ since the order of execution is irrelevant if $s = 1$. On the other hand, $0 \nVdash_\mathtt{s} \mathtt{emit} \diamond \mathtt{pres}$ because in state 0 both methods are not confluent. Specifically, $0.\mathtt{pres}(\_) = 0 \neq 1 = (0 \odot \mathtt{emit}(\_)).\mathtt{pres}(\_)$. □

A special case are *linear precedence policies* where $\mu \Vdash_\mathtt{c} \downarrow m$ for all $m \in \mathsf{M}_\mathtt{c}$ and $\mu \Vdash_\mathtt{c} m \to n$ is a linear ordering on $\mathsf{M}_\mathtt{c}$, for all policy states μ. Then, for no state we have $\mu \Vdash_\mathtt{c} m_1 \diamond m_2$, so there is no concurrency and thus no confluence requirement to satisfy at all. Coherence of $\mathtt{c}$ is trivially satisfied whatever the semantics of method calls. For any two admissible methods one takes precedence over the other and thus the enabling relation becomes deterministic. There is however a risk of deadlock which can be excluded if we assume that threads always call methods in order of decreasing precedence.

The other extreme case is where the policy makes all methods concurrently enabled, i.e., $\mu \Vdash_{\mathsf{c}} m_1 \diamond m_2$ for all policy states μ and methods m_1, m_2. This avoids deadlock completely and gives maximal concurrency but imposes the strongest confluence condition, viz. independently of the scheduling order of any two methods, the resulting variable state must be the same. This requires complete isolation of the effects of any two methods. Such an extreme is used, e. g., in the CR library [19]. The typical CSM variable, however, will strike a trade-off between these two extremes. It will impose a sensible set of precedences that are strong enough to ensure coherent implementations and thus determinacy for policy-conformant scheduling, while at the same time being sufficiently relaxed to permit concurrent implementations and avoiding unnecessary deadlocks risking that programs are rejected by the compiler as un-scheduleable.

Whatever the policies, if the variables are coherent, then all policy-conformant interleavings are indistinguishable for each CSM variable. To state schedule invariance in its general form we lift method actions and independence to multi-variable sequences of methods calls $\mathsf{A} = \{\mathsf{c}.m(v) \mid \mathsf{c} \in \mathsf{O}, m(v) \in \mathsf{A}_{\mathsf{c}}\}$. For a given sequence $\alpha \in \mathsf{A}^*$ let $\pi_{\mathsf{c}}(\alpha) \in \mathsf{A}_{\mathsf{c}}^*$ be the projection of α on c, formally $\pi_{\mathsf{c}}(\varepsilon) = \varepsilon$, $\pi_{\mathsf{c}}(\mathsf{c}.m(v)\,\alpha) = m(v)\,\pi_{\mathsf{c}}(\alpha)$ and $\pi_{\mathsf{c}}(\mathsf{c}'.m(v)\,\alpha) = \pi_{\mathsf{c}}(\alpha)$ for $\mathsf{c}' \neq \mathsf{c}$. A global memory $\Sigma \in \mathbb{S} = \prod_{\mathsf{c}\in\mathsf{O}} \mathbb{S}_{\mathsf{c}}$ assigns a local memory $\Sigma.\mathsf{c} \in \mathbb{S}_{\mathsf{c}}$ to each variable c. We write *init* for the initial memory that has $\mathit{init}.\mathsf{c} = \mathit{init}_{\mathsf{c}}$ and $(\mathit{init}.\mathsf{c})^{\#} = \varepsilon \in \mathbb{P}_{\mathsf{c}}$.

Given a global memory $\Sigma \in \mathbb{S}$ and sequences $\alpha, \beta \in \mathsf{A}^*$ of method calls, we extend the independence relation of Definition 4 variable-wise, defining $\Sigma \Vdash \alpha \diamond \beta$ iff $(\Sigma.\mathsf{c})^{\#} \Vdash_{\mathsf{c}} (\pi_{\mathsf{c}}(\alpha))^{\#} \diamond (\pi_{\mathsf{c}}(\beta))^{\#}$. The application of a method call $a \in \mathsf{A}$ to a memory $\Sigma \in \mathbb{S}$ is written $\Sigma.a \in \mathbb{S}$ and defined $(\Sigma.(\mathsf{c}.m(v))).\mathsf{c} = (\Sigma.\mathsf{c}).m(v)$ and $(\Sigma.(\mathsf{c}.m(v))).\mathsf{c}' = \Sigma.\mathsf{c}'$ for $\mathsf{c}' \neq \mathsf{c}$. Analogously, method actions are lifted to global memories, i.e., $(\Sigma \odot \mathsf{c}.m(v)).\mathsf{c}' = \Sigma.\mathsf{c}'$ if $\mathsf{c}' \neq \mathsf{c}$ and $(\Sigma \odot \mathsf{c}.m(v)).\mathsf{c} = \Sigma.\mathsf{c} \odot m(v)$.

Proposition 1 (Commutation). *Let all* CSM *variables be policy-coherent and* $\Sigma \Vdash a \diamond \alpha$ *for a memory* $\Sigma \in \mathbb{S}$*, method call* $a \in V$ *and sequences of method calls* $\alpha \in V^*$*. Then,* $\Sigma \odot a \odot \alpha = \Sigma \odot \alpha \odot a$ *and* $\Sigma.a = (\Sigma \odot \alpha).a$.

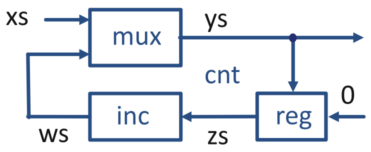
2.6 Policies and Modularity

Consider the synchronous data-flow network `cnt` in Fig. 3b with three process nodes, a multiplexer `mux`, a register `reg` and an incrementor `inc`. Their DCoL code is given in Fig. 3a. The network implements a settable counter, which produces at its output `ys` a stream of consecutive integers, incremented with each clock tick. The wires `ys`, `zs` and `ws` are IVars (see Example 2) carrying a single integer value per tick. The input `xs` is a pure Esterel signal (see Example 1). The counter state is stored by `reg` in a local variable `xv` with `read` and `write` methods that can be called by a single thread only. The register is initialised to value 0 and in each subsequent tick the value at `ys` is stored. The `inc` takes the value at `zs` and increments it. When the signal `xs` is absent, `mux` passes the incremented value on ws to ys for the next tick. Otherwise, if xs is present then mux resets ys.

The evaluation order is implemented by the policies of the IVars ys, zs and ws. In each case the put method takes precedence over get which makes sure that the latter is blocked until the former has been executed. The causality cycle of the feedback loop is broken by the fact that the reg node first sends the current counter value to zs before it waits for the new value at ys. The other nodes mux and inc, in contrast, first read their inputs and then send to their output.

```
module cnt
[ % mux node
  loop
    v = xs.pres();
    if v then ys.put(0);
           else u = ws.get();
                ys.put(u);
  end
] ||
[ % reg node
  xv.write(0);
  loop
    v = xv.read(); zs.send(v);
    u = ys.get(); xv.write(u);
  end
] ||
[ % inc node
  loop
    v = zs.get(); ws.put(v+1);
  end
]
```

(a) Network with mux, reg, inc threads.



(b) Block diagram of the feedback network.

```
module cnt-cmp
reg.init(0);
[ % mux-cmp node
  loop
    v = xs.pres();
    if v then reg.set(0);
           else u = ws.get();
                reg.set(u);
  end
] ||
[ % inc-cmp node
  loop
    v = reg.get(); ws.put(v+1);
  end
]
```

(c) Network with reg as a precompiled DCoL object.

Fig. 3. Synchronous data-flow network cnt built from control-flow processes.

Now suppose, for modularity, the reg node is pre-compiled into a synchronous IO automaton to be used by mux and inc as a black box component. Then, reg must be split into three *modes* [20] reg.init, reg.get and reg.set that can be called independently in each instant. The init mode initialises the register memory with 0. The get mode extracts the buffered value and set stores a new value into the register. Since there may be data races if get and set are called concurrently on reg, a policy must be imposed. In the scheduling of Fig. 3b, first reg.get is executed to place the output on zs. Then, reg waits for mux to produce the next value of ys from xs or ws. Finally, reg.set is executed to store the current value of ys for the next tick. Thus, the natural policy for the register is to require that in each tick `set` is called by at most one thread and if so no concurrent call to `get` by another thread happens afterwards. In addition, the policy requires `init` to take place at least once before any `set` or `get`. Hence, the policy has two states $\mathbb{P}_{\texttt{reg}} = \{0, 1\}$ with initial $\varepsilon = 0$ and admissibility such that $0 \Vdash_{\texttt{reg}} \downarrow m$ iff $m = \texttt{init}$ and $1 \Vdash_{\texttt{reg}} \downarrow m$ for all m. The transitions are $0 \odot \texttt{init} = 1$ and $1 \odot m = 1$ for all $m \in \mathsf{M}_{\texttt{reg}}$. Further, for coherence, in state 1 no `set` may be concurrent and every `get` must take place before any concurrent `set`. This means, we have $1 \Vdash_{\texttt{reg}} m \rightarrow \texttt{set}$ for all $m \in \{\texttt{get}, \texttt{set}\}$. Figure 3c shows the partially compiled code in which `reg` is treated as a compiled object. The policy on `reg` makes sure the accesses by `mux` and `inc` are scheduled in the right way (see Example 4). Note that `reg` is not an IVar because it has memory.

The `cnt` example exhibits a general pattern found in the modular compilation of SP: Modules (here `reg`) may be exercised *several times* in a synchronous tick through *modes* which are executed in a specific *prescribed order*. Mode calls (here `reg.set`, `reg.get`) in the same module are coupled via common *shared memory* (here the local variable `xs`) while mode calls in distinct modules are isolated from each other [15,20].

3 Constructive Semantics of DCoL

To formalise our semantics it is technically expedient to keep track of the *completion status* of each active thread inside the program expression. This results in a syntax for *processes* distinguished from programs in that each parallel composition $P_1 \,{}_{k_1}\Vert_{k_2}\, P_2$ is labelled by *completion codes* $k_i \in \{\bot, 0, 1\}$ which indicate whether each thread is *waiting* $k_i = \bot$, *terminated* 0 or *pausing* $k_i = 1$. Since we remove a process from the parallel as soon as it terminates then the code $k_i = 0$ cannot occur. An expression $P_1 \Vert P_2$ is considered a special case of a process with $k_i = \bot$. The formal semantics is given by a reduction relation on processes

$$\Sigma; \Pi \vdash P \overset{\boldsymbol{m}}{\Rightarrow} \Sigma' \vdash_{k'} P' \qquad (2)$$

specified by the inductive rules in Figs. 4 and 5. The relation (2) determines an instantaneous *sequential reduction step* of process P, called an *sstep*, that follows a sequence of enabled method calls $\boldsymbol{m} \in \mathsf{M}^*$ in sequential program order in P. This does not include any context switches between concurrent threads inside P. For thread communication, several ssteps must be chained up, as described later. The sstep (2) results in an updated memory Σ' and residual process P'. The subscript k' is a completion code, described below. The reduction (2) is performed in a context consisting of a global memory $\Sigma \in \mathbb{S}$ (*must* context) containing the current state of all CSM variables and an environment prediction $\Pi \subseteq \mathsf{M}^*$ (*can* context). The prediction records all potentially outstanding methods sequences from threads running *concurrently* with P.

We write $\pi_{\mathsf{c}}(\boldsymbol{m}) \in \mathsf{M}^*_{\mathsf{c}}$ for the projection of a method sequence $\boldsymbol{m} \in \mathsf{M}^*$ to variable c and write $\pi_{\mathsf{c}}(\Pi)$ for its lifting to sets of sequences. Prefixing is lifted, too, i.e., $\mathsf{c}.m \odot \Pi = \{\mathsf{c}.m\, \boldsymbol{m} \mid \boldsymbol{m} \in \Pi\}$ for any $\mathsf{c}.m \in \mathsf{M}$.

Performing a method call $\mathsf{c}.m(v)$ in $\Sigma; \Pi$ advances the *must* context to $\Sigma \odot \mathsf{c}.m(v)$ but leaves Π unchanged. The sequence of methods $\boldsymbol{m} \in \mathsf{M}^*$ in (2) is *enabled* in $\Sigma; \Pi$, written $[\Sigma, \Pi] \Vdash \downarrow \boldsymbol{m}$ meaning that $[(\Sigma.\mathsf{c})^\#, \pi_\mathsf{c}(\Pi)] \Vdash_\mathsf{c} \downarrow \pi_\mathsf{c}(\boldsymbol{m})$ for all $\mathsf{c} \in \mathsf{O}$. In this way, the context $[\Sigma, \Pi]$ forms a joint policy state for all variables for the TUC P, in the sense of Sect. 2 (Definition 3).

Sequence

$$\frac{\Sigma; \Pi \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P' \qquad k' \neq 0}{\Sigma; \Pi \vdash P \,;\, Q \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P' \,;\, Q}\ \mathsf{Seq}_1$$

$$\frac{\Sigma; \Pi \vdash P \overset{\boldsymbol{m}_1}{\Longrightarrow} \Sigma' \vdash_0 P' \qquad \Sigma'; \Pi \vdash Q \overset{\boldsymbol{m}_2}{\Longrightarrow} \Sigma'' \vdash_{k'} Q'}{\Sigma; \Pi \vdash P \,;\, Q \overset{\boldsymbol{m}_1 \boldsymbol{m}_2}{\Longrightarrow} \Sigma'' \vdash_{k'} Q'}\ \mathsf{Seq}_2$$

Completion

$$\frac{}{\Sigma; \Pi \vdash \mathtt{skip} \overset{\varepsilon}{\Rightarrow} \Sigma \vdash_0 \mathtt{skip}}\ \mathsf{Cmp}_1 \qquad \frac{}{\Sigma; \Pi \vdash \mathtt{pause} \overset{\varepsilon}{\Rightarrow} \Sigma \vdash_1 \mathtt{pause}}\ \mathsf{Cmp}_2$$

Recursion

$$\frac{\Sigma; \Pi \vdash P\{\mathsf{rec}\, p.\, P/p\} \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'}{\Sigma; \Pi \vdash \mathsf{rec}\, p.\, P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'}\ \mathsf{Rec}$$

Fig. 4. SStep reductions for sequence, completion and recursion.

Most of the rules in Figs. 4 and 5 should be straightforward for the reader familiar with structural operational semantics. Seq_1 is the case of a sequential $P; Q$ where P pauses or waits ($k' \neq 0$) and Seq_2 is where P terminates and control passes into Q. The statements `skip` and `pause` are handled by rules Cmp_1 and Cmp_2. The rule Rec explains recursion $\mathsf{rec}\, p.P$ by syntactic unfolding of the recursion body P. All interaction with the memory takes place in the method calls $\mathtt{let}\, x = \mathsf{c}.m(e)\ \mathtt{in}\, P$. Rule Let_1 is applicable when the method call is enabled, i.e., $[\Sigma, \Pi] \Vdash \downarrow \mathsf{c}.m$. Since processes are closed, the argument expression e must evaluate, $eval(e) = v$, and we obtain the new memory $\Sigma \odot \mathsf{c}.m(v)$ and return value $\Sigma.\mathsf{c}.m(v)$. The return value is substituted for the local (stack allocated) identifier x, giving the continuation process $P\{\Sigma.\mathsf{c}.m(v)/x\}$ which is run in the updated context $\Sigma \odot \mathsf{c}.m(v); \Pi$. The prediction Π remains the same. The second rule Let_2 is used when the method call is blocked or the thread wants to wait and yield to the scheduler. The rules for conditionals Cnd_1, Cnd_2 are canonical. More interesting are the rules Par_1–Par_4 for parallel composition, which implement non-deterministic thread switching. It is here where we need to generate predictions and pass them between the threads to exercise the policy control.

The key operation is the computation of the *can*-prediction of a process P to obtain an over-approximation of the set of possible method sequences potentially executed by P. For compositionality we work with sequences $can^s(P) \subseteq \mathsf{M}^* \times \{0, 1\}$ *stoppered* with a completion code 0 if the sequence ends in termination or

Method Call

$$\frac{[\Sigma, \Pi] \Vdash {\downarrow}\mathsf{c}.m \quad eval(e) = v \quad \Sigma \odot \mathsf{c}.m(v); \Pi \vdash P\{\Sigma.\mathsf{c}.m(v)/x\} \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'}{\Sigma; \Pi \vdash \mathtt{let}\, x = \mathsf{c}.m(e)\ \mathtt{in}\, P \xRightarrow{\mathsf{c}.m\, \boldsymbol{m}} \Sigma' \vdash_{k'} P'}\ \mathsf{Let}_1$$

$$\frac{}{\Sigma; \Pi \vdash \mathtt{let}\, x = \mathsf{c}.m(e)\ \mathtt{in}\, P \overset{\varepsilon}{\Rightarrow} \Sigma \vdash_{\bot} \mathtt{let}\, x = \mathsf{c}.m(e)\, \mathtt{in}\, P}\ \mathsf{Let}_2$$

Conditional

$$\frac{eval(e) = \mathsf{true} \quad \Sigma; \Pi \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'}{\Sigma; \Pi \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'}\ \mathsf{Cnd}_1$$

$$\frac{eval(e) = \mathsf{false} \quad \Sigma; \Pi \vdash Q \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} Q'}{\Sigma; \Pi \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} Q'}\ \mathsf{Cnd}_2$$

Parallel

$$\frac{\Sigma; \Pi \otimes can(Q) \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P' \quad k' \neq 0}{\Sigma; \Pi \vdash P\ {}_{k}\|_{k_Q}\ Q \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k' \sqcap k_Q} P'\ {}_{k'}\|_{k_Q}\ Q}\ \mathsf{Par}_1$$

$$\frac{\Sigma; \Pi \otimes can(Q) \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_0 P'}{\Sigma; \Pi \vdash P\ {}_{k}\|_{k_Q}\ Q \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k_Q} Q}\ \mathsf{Par}_2$$

$$\frac{\Sigma; \Pi \otimes can(P) \vdash Q \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} Q' \quad k' \neq 0}{\Sigma; \Pi \vdash P\ {}_{k_P}\|_{k}\ Q \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k_P \sqcap k'} P\ {}_{k_P}\|_{k'}\ Q'}\ \mathsf{Par}_3$$

$$\frac{\Sigma; \Pi \otimes can(P) \vdash Q \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_0 Q'}{\Sigma; \Pi \vdash P\ {}_{k_P}\|_{k}\ Q \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k_P} P}\ \mathsf{Par}_4$$

Fig. 5. SStep reductions for method calls, conditional and parallel.

1 if it ends in pausing. The symbols $\bot_0$, $\bot_1$ and $\top$ are the *terminated*, *paused* and *fully unconstrained can* contexts, respectively, with $\bot_0 = \{(\varepsilon, 0)\}$, $\bot_1 = \{(\varepsilon, 1)\}$ and $\top = \mathsf{M}^* \times \{0, 1\}$. The set $can^s(P)$, defined in Fig. 6, is extracted from the structure of P using prefixing $\mathsf{c}.m \odot \Pi'$, choice $\Pi_1' \oplus \Pi_2' = \Pi_1' \cup \Pi_2'$, parallel $\Pi_1' \otimes \Pi_2'$ and sequential composition $\Pi_1' \cdot \Pi_2'$. Sequential composition is obtained pairwise on stoppered sequences such that $(\boldsymbol{m}, 0) \cdot (\boldsymbol{n}, c) = (\boldsymbol{m}\, \boldsymbol{n}, c)$ and $(\boldsymbol{m}, 1) \cdot (\boldsymbol{n}, c) = (\boldsymbol{m}, 1)$. As a consequence, $\bot_0 \cdot \Pi' = \Pi'$ and $\bot_1 \cdot \Pi' = \bot_1$. Parallel composition is pairwise free interleaving with synchronisation on completion codes. Specifically, a product $(\boldsymbol{m}, c) \otimes (\boldsymbol{n}, d)$ generates all interleavings of $\boldsymbol{m}$ and $\boldsymbol{n}$ with a completion that models a parallel composition that terminates iff both threads terminate and pauses if one pauses. Formally, $(\boldsymbol{m}, c) \otimes (\boldsymbol{n}, d) = \{(\boldsymbol{c}, max(c, d)) \mid \boldsymbol{c} \in \boldsymbol{m} \otimes \boldsymbol{n}\}$. Thus, $\Pi_P' \otimes \Pi_Q' = \bot_0$ iff $\Pi_P' = \bot_0 = \Pi_Q'$ and $\Pi_P' \otimes \Pi_Q' = \bot_1$ if $\Pi_P' = \bot_1 = \Pi_Q'$, or $\Pi_P' = \bot_0$ and $\Pi_Q' = \bot_1$, or $\Pi_P' = \bot_1$ and $\Pi_Q' = \bot_0$. From $can^s(P)$ we obtain $can(P) \subseteq \mathsf{M}^*$ by dropping all stopper codes, i.e., $can(P) = \{\boldsymbol{m} \mid \exists d.\, (\boldsymbol{m}, d) \in can^s(P)\}$.

The rule Par_1 exercises a parallel $P\ {}_{k}\|_{k_Q}\ Q$ by performing an sstep in P. This sstep is taken in the extended context $\Sigma; \Pi \otimes can(Q)$ in which the prediction of the sibling Q is added to the method prediction Π for the outer environment

$$
\begin{aligned}
can^s(\mathtt{skip}) &= can^s(p) = \bot_0 \qquad can^s(\mathtt{pause}) = \bot_1 \\
can^s(\mathsf{rec}\, p.\, P) &= can^s(P) \qquad can^s(P \parallel Q) = can^s(P) \otimes can^s(Q) \\
can^s(P\,;\,Q) &= \begin{cases} can^s(P) & \text{if } can^s(P) \subseteq \mathsf{M}^* \times \{1\} \\ can^s(P) \cdot can^s(Q) & \text{otherwise} \end{cases} \\
can^s(\mathtt{let}\, x = \mathtt{c}.m(e)\, \mathtt{in}\, P) &= \mathtt{c}.m \odot can^s(P) \\
can^s(\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q) &= \begin{cases} can^s(P) & \text{if } eval(e) = \mathsf{true} \\ can^s(Q) & \text{if } eval(e) = \mathsf{false} \\ can^s(P) \oplus can^s(Q) & \text{otherwise.} \end{cases}
\end{aligned}
$$

Fig. 6. Computing the *can* prediction.

in which the parent $P \parallel Q$ is running. In this way, Q can block method calls of P. When P finally yields as P' with a non-terminating completion code, $0 \neq k' \in \{\bot, 1\}$, the parallel completes as $P' \,{}_{k'}\!\parallel_{k_Q}\, Q$ with code $k' \sqcap k_Q$. This operation is defined $k_1 \sqcap k_2 = 1$ if $k_1 = 1 = k_2$ and $k_1 \sqcap k_2 = \bot$, otherwise. When P terminates its sstep with code $k' = 0$ then we need rule Par_2 which removes child P' from the parallel composition. The rules $\mathsf{Par}_3, \mathsf{Par}_4$ are symmetrical to $\mathsf{Par}_1, \mathsf{Par}_2$. They run the right child Q of a parallel $P \,{}_{k_P}\!\parallel_{k}\, Q$.

Completion and Stability. A process P' is 0-*stable* if $P' = \mathtt{skip}$ and 1-*stable* if $P' = \mathtt{pause}$ or $P' = P'_1\,;\,P'_2$ and P'_1 is 1-*stable*, or $P' = P'_1 \,{}_1\!\parallel_1\, P'_2$, and P'_i are 1-stable. A process is *stable* if it is 0-stable or 1-stable. A process expression is *well-formed* if in each sub-expression $P_1 \,{}_{k_1}\!\parallel_{k_2}\, P_2$ of P the completion annotations are matching with the processes, i.e., if $k_i \neq \bot$ then P_i is k_i-stable. Stable processes are well-formed by definition. For stable processes we define a *(syntactic) tick function* which steps a stable process to the next tick. It is defined such that $\sigma(\mathtt{skip}) = \mathtt{skip}$, $\sigma(\mathtt{pause}) = \mathtt{skip}$, $\sigma(P'_1\,;\,P'_2) = \sigma(P'_1)\,;\,P'_2$ and $\sigma(P'_1 \,{}_{k_1}\!\parallel_{k_2}\, P'_2) = \sigma(P'_1) \parallel \sigma(P'_2)$.

Example 4. The data-flow `cnt-cmp` from Fig. 3c can be represented as a DCoL process in the form $C = \mathtt{reg.init}(0); (M \,{}_\bot\!\parallel_\bot\, I)$ with

$$
\begin{aligned}
M &=_{df} \mathsf{rec}\, p.\, \mathtt{v} = \mathtt{xs.pres}(); P(v); \mathtt{pause}; p \\
P(v) &=_{df} \mathtt{if}\ v\ \mathtt{then}\ \mathtt{reg.set}(0); \mathtt{else}\ Q \\
Q &=_{df} u = \mathtt{ws.get}(); \mathtt{reg.set}(u); \\
I &=_{df} \mathsf{rec}\, q.\, \mathtt{v} = \mathtt{reg.get}(); \mathtt{ws.put}(v+1); \mathtt{pause}; q.
\end{aligned}
$$

Let us evaluate process C from an initialised memory Σ_0 such that $\Sigma_0.\mathtt{xs} = 0 = \Sigma_0.\mathtt{ws}$, and empty environment prediction $\{\epsilon\}$.

The first sstep is executed from the context $\Sigma_0; \{\epsilon\}$ with empty *can* prediction. Note that $\mathtt{reg.init}(0); (M \,{}_\bot\!\parallel_\bot\, I)$ abbreviates $\mathtt{let}\ \_ = \mathtt{reg.init}(0)\ \mathtt{in}$ $(M \,{}_\bot\!\parallel_\bot\, I)$. In context $\Sigma_0; \{\epsilon\}$ the method call $\mathtt{reg.init}(0)$ is enabled, i.e., $[\Sigma_0, \{\epsilon\}] \Vdash \downarrow \mathtt{reg.init}$. Since $eval(0) = 0$, we can execute the first method call of C using rule Let_1. This advances the memory to $\Sigma_1 = \Sigma_0 \odot \mathtt{reg.init}(0)$.

The continuation process $M \;{}_{\perp}\|_{\perp}\; I$ is now executed in context $\Sigma_1; \perp_0$. The left child M starts with method call $\texttt{xs.pres}()$ and the right child I with $\texttt{reg.get}()$. The latter is admissible, since $(\Sigma_1.\texttt{reg})^{\#} = 1$. Moreover, $\texttt{get}$ does not need to honour any precedences, whence it is enabled, $[\Sigma_1, \Pi] \Vdash {\downarrow}\texttt{reg.get}$ for any Π. On the other hand, $\texttt{xs.pres}$ in M is enabled only if $(\Sigma_1.\texttt{xs})^{\#} = 1$ or if there is no concurrent $\texttt{emit}$ predicted for $\texttt{xs}$. Indeed, this is the case: The concurrent context of M is $\Pi_I = \{\epsilon\} \otimes \mathit{can}(I) = \mathit{can}(I) = \{\texttt{reg.get} \cdot \texttt{ws.put}\}$. We project $\pi_{\texttt{xs}}(\Pi_I) = \{\epsilon\}$ and find $[\Sigma_1, \Pi_I] \Vdash {\downarrow}\texttt{xs.pres}$. Hence, we have a non-deterministic choice to take an sstep in M or in I. Let us use rule $\mathsf{Par}_1/\mathsf{Par}_2$ to run M in context $\Sigma; \Pi_I$. By loop unfolding Rec and rule Let_1 we execute the present test of M which returns the value $\Sigma_1.\texttt{xs.pres}() = \texttt{false}$. This leads to an updated memory $\Sigma_2 = \Sigma_1 \odot \texttt{xs.pres}() = \Sigma_1$ and continuation process $P(\texttt{false}); \texttt{pause}; M$. In this (right associated) sequential composition we first execute $P(\texttt{false})$ where the conditional rule Cnd_2 switches to the $\texttt{else}$ branch Q which is $u = \texttt{ws.get}(); \texttt{reg.set}(u);$, still in the context Σ_2, Π_I. The reading of the data-flow variable $\texttt{ws}$, however, is not enabled, $[\Sigma_2, \Pi_I] \nVdash {\downarrow}\texttt{ws.get}$, because $(\Sigma_2.\texttt{ws})^{\#} = 0$ and thus $\texttt{get}$ not admissible. The sstep blocks with rule Let_2:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_2; \Pi_I \vdash Q \overset{\epsilon}{\Rightarrow} \Sigma_2 \vdash_{\perp} Q}\;\mathsf{Let}_2}{\Sigma_2; \Pi_I \vdash P(\texttt{false}) \overset{\epsilon}{\Rightarrow} \Sigma_2 \vdash_{\perp} Q}\;\mathsf{Cnd}_2}{\Sigma_2; \Pi_I \vdash P(\texttt{false}); \texttt{pause}; M \overset{\epsilon}{\Rightarrow} \Sigma_2 \vdash_{\perp} Q; \texttt{pause}; M}\;\mathsf{Seq}_1}{\Sigma_1; \Pi_I \vdash \texttt{v} = \texttt{xs.pres}(); P(v); \texttt{pause}; M \overset{\epsilon}{\Rightarrow} \Sigma_2 \vdash_{\perp} Q; \texttt{pause}; M}\;\mathsf{Let}_1(\Sigma_1; \Pi_I \vdash {\downarrow}\texttt{xs.pres})}{\Sigma_1; \Pi_I \vdash M \overset{m_2}{\Longrightarrow} \Sigma_2 \vdash_{\perp} Q; \texttt{pause}; M}\;\mathsf{Rec}}{\Sigma_1; \{\epsilon\} \vdash M \;{}_{\perp}\|_{\perp}\; I \overset{m_2}{\Longrightarrow} \Sigma_2 \vdash_{\perp} (Q; \texttt{pause}; M) \;{}_{\perp}\|_{\perp}\; I}\;\mathsf{Par}_1}{\Sigma; \{\epsilon\} \vdash C \overset{m_1 m_2}{\Longrightarrow} \Sigma_2 \vdash_{\perp} (Q; \texttt{pause}; M) \;{}_{\perp}\|_{\perp}\; I}\;\mathsf{Let}_1(\Sigma; \perp_0 \Vdash {\downarrow}\texttt{reg.init})$$

where $m_1 = \texttt{reg.init}$ and $m_2 = \texttt{xs.pres}$. In the next sstep, from $\Sigma_2; \Pi_Q$ with $\Pi_Q = \{\epsilon\} \otimes \mathit{can}(Q; \texttt{pause}; M) = \mathit{can}(Q; \texttt{pause}; M) = \{\texttt{ws.get} \cdot \texttt{reg.set}\}$ we let the process I execute its $\texttt{reg.get}()$ with rules Rec and Let_1. The return value is $v = \Sigma_2.\texttt{reg.get}() = 0$. Then, from the updated memory $\Sigma_3 = \Sigma_2 \odot \texttt{reg.get}()$ we run the continuation process $\texttt{ws.put}(0+1); \texttt{pause}; I$. The $\texttt{ws.put}$ is enabled if the IVar is empty and there is no concurrent $\texttt{put}$ on $\texttt{ws}$ predicted from M. Both conditions hold since $(\Sigma_3.\texttt{ws})^{\#} = (\Sigma.\texttt{ws})^{\#} = 0$ and $\pi_{\texttt{ws}}(\Pi_Q) = \{\texttt{get}\}$. Therefore, $[\Sigma_3, \Pi_Q] \Vdash {\downarrow}\texttt{ws.put}$. With the evaluation $\mathit{eval}(0+1) = 1$ the rule Let_1 permits us to update the memory as $\Sigma_4 = \Sigma_3 \odot \texttt{ws.put}(1)$ and continue with process $\texttt{pause}; I$ which completes by pausing. Formally, this sstep is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_4; \Pi_Q \vdash \texttt{pause} \overset{\epsilon}{\Rightarrow} \Sigma_4 \vdash_1 \texttt{pause}}\;\mathsf{Cmp}_2}{\Sigma_4; \Pi_Q \vdash \texttt{pause}; I \overset{\epsilon}{\Rightarrow} \Sigma_4 \vdash_1 \texttt{pause}; I}\;\mathsf{Seq}_1}{\Sigma_3; \Pi_Q \vdash \texttt{ws.put}(0+1); \texttt{pause}; I \overset{m_4}{\Longrightarrow} \Sigma_4 \vdash_1 \texttt{pause}; I}\;\mathsf{Let}_2}{\Sigma_2; \Pi_Q \vdash v = \texttt{reg.get}(); \texttt{ws.put}(v+1); \texttt{pause}; I \overset{m_3 m_4}{\Longrightarrow} \Sigma_4 \vdash_1 \texttt{pause}; I}\;\mathsf{Let}_1}{\Sigma_2; \Pi_Q \vdash I \overset{m_3 m_4}{\Longrightarrow} \Sigma_4 \vdash_1 \texttt{pause}; I}\;\mathsf{Rec}}{\Sigma_2; \{\epsilon\} \vdash (Q; \texttt{pause}; M) \;{}_{\perp}\|_{\perp}\; I \overset{m_3 m_4}{\Longrightarrow} \Sigma_4 \vdash_{\perp} (Q; \texttt{pause}; M) \;{}_{\perp}\|_{1}\; (\texttt{pause}; I)}\;\mathsf{Par}_3$$

where $m_3 = \texttt{reg.get}$ and $m_4 = \texttt{ws.put}$. In the next sstep the waiting method $u = \texttt{ws.get}$ in Q is now admissible and can proceed, $(\Sigma_4.\texttt{ws})^{\#} = ((\Sigma_3 \odot \texttt{ws.put}(1)).\texttt{ws})^{\#} = 1$ and thus $[\Sigma_4, \Pi] \Vdash \downarrow \texttt{ws.get}$ for all Π. The return value is $u = \Sigma_4.\texttt{ws.get}() = 1$, the updated memory $\Sigma_5 = \Sigma_4 \odot \texttt{ws.put}(1)$ and the continuation process $\texttt{reg.set}(1); \texttt{pause}; M$. The register $\texttt{set}$ method is admissible since $(\Sigma_4.\texttt{reg})^{\#} = 1$ and also enabled because there is no $\texttt{get}$ predicted in the concurrent environment $\bot_0$. Thus, $[\Sigma_5, \bot_0] \Vdash \downarrow \texttt{reg.set}$. The execution of the method yields the memory $\Sigma_6 = \Sigma_5 \odot \texttt{reg.set}(1)$ with continuation process $\texttt{pause}$; M which completes by pausing. This yields the derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_6; \{\epsilon\} \vdash \texttt{pause}; M \overset{\epsilon}{\Rightarrow} \Sigma_6 \vdash_1 \texttt{pause}; M}\,\mathsf{Cmp}_2}{\Sigma_5; \{\epsilon\} \vdash \texttt{reg.set}(1); \texttt{pause}; M \xRightarrow{m_6} \Sigma_6 \vdash_1 \texttt{pause}; M}\,\mathsf{Let}_1}{\Sigma_4; \{\epsilon\} \vdash Q; \texttt{pause}; M \xRightarrow{m_5 m_6} \Sigma_6 \vdash_1 \texttt{pause}; M}\,\mathsf{Let}_1}{\Sigma_4; \{\epsilon\} \vdash (Q; \texttt{pause}; M)\ {}_{\bot}\Vert_1\ (\texttt{pause}; I) \xRightarrow{m_5 m_6} \Sigma_6 \vdash_1 (\texttt{pause}; M)\ {}_1\Vert_1\ (\texttt{pause}; I)}\,\mathsf{Par}_2$$

where $m_5 = \texttt{ws.get}$ and $m_6 = \texttt{reg.set}$. To justify the rule Par_2 consider that $\{\epsilon\} \otimes \mathit{can}(\texttt{pause}; I) = \{\epsilon\} \otimes \{\epsilon\} = \{\epsilon\}$. At this point we have reached a 1-stable process. With the tick function we advance to the next tick, $\sigma((\texttt{pause}; M)\ {}_1\Vert_1\ (\texttt{pause}; I)) = (\texttt{skip}; M)\ {}_{\bot}\Vert_{\bot}\ (\texttt{skip}; I)$ which behaves like $M\ {}_{\bot}\Vert_{\bot}\ I$. $\square$

3.1 Determinacy, Termination and Constructiveness

Determinacy of DCoL is a result of two components, monotonicity of policy-conformant scheduling and CSM coherence. Monotonicity ensures that whenever a method is executable and policy-enabled, then it remains policy-enabled under arbitrary ssteps of the environment. Symmetrically, the environment cannot be blocked by a thread taking policy-enabled computation steps.

The second building block for determinacy is CSM variable coherence. Consider a context $\Sigma; \Pi_Q$ in which we run an sstep of P with prediction Π_Q for concurrent process Q, resulting in a final memory Σ'_P arising from executing a sequence $\boldsymbol{m}_P$ of method calls from P. Because of the policy constraint, the sequence $\boldsymbol{m}_P$ must be enabled under all predictions $\boldsymbol{n} \in \Pi_Q$, i.e., $[\Sigma, \boldsymbol{n}] \Vdash \downarrow \boldsymbol{m}_P$. Suppose, on the other side, we sstep the process Q in the same memory Σ with prediction Π_P for P, resulting in an action sequence $\boldsymbol{m}_Q$ and final memory Σ'_Q. Then, by the same reasoning, $[\Sigma, \boldsymbol{n}] \Vdash \downarrow \boldsymbol{m}_Q$ for all $\boldsymbol{n} \in \Pi_P$. But since $\boldsymbol{m}_P$ is an actual execution of P it must be in the prediction for P, i.e., $\boldsymbol{m}_P \in \Pi_P$ and symmetrically, $\boldsymbol{m}_Q \in \Pi_Q$. But then we have $[\Sigma, \boldsymbol{m}_Q] \Vdash \downarrow \boldsymbol{m}_P$ and $[\Sigma, \boldsymbol{m}_P] \Vdash \downarrow \boldsymbol{m}_P$ which means $\Sigma \Vdash \boldsymbol{m}_P \diamond \boldsymbol{m}_Q$. Now if the semantics of method calls is policy-coherent then the Monotonicity can be exploited to derive a confluence property for processes which guarantees that $\boldsymbol{m}_P$ can still be executed by P in state Σ'_Q and $\boldsymbol{m}_Q$ by Q in state Σ'_P, and both lead to the same final memory.

Theorem 1 (Diamond Property). *If all* CSM *variables are policy-coherent then the sstep semantics is confluent. Formally, given two derivations* $\Sigma; \Pi \vdash P \overset{\boldsymbol{m}_1}{\Longrightarrow} \Sigma_1 \vdash_{k_1} P_1$ *and* $\Sigma; \Pi \vdash P \overset{\boldsymbol{m}_2}{\Longrightarrow} \Sigma_2 \vdash_{k_2} P_2$*, Then, there exist* Σ'*,* k' *and* P' *such that* $\Sigma_1; \Pi \vdash P_1 \overset{\boldsymbol{n}_1}{\Longrightarrow} \Sigma' \vdash_{k'} P'$ *and* $\Sigma_1; \Pi \vdash P_2 \overset{\boldsymbol{n}_2}{\Longrightarrow} \Sigma' \vdash_{k'} P'$*.*

Theorem 1 shows that no matter how we schedule the ssteps of local threads to create an sstep of a parallel composition, the final result will not diverge. This does not guarantee completion of a process. However, it implies that the question of whether P blocks or makes progress does not depend on the order in which concurrent threads are scheduled. Either a process completes or it does not. All ssteps in a process can be scheduled with maximal parallelism without interference.

A main program P is run at the top level in an "environmentally closed" form of ssteps (2) where the prediction is empty $\Pi = \{\epsilon\}$ and thus acts neutrally. We iterate such ssteps to construct a macro-step reaction. Let us write

$$\Sigma \vdash P \Rightarrow \Sigma' \vdash P' \tag{3}$$

if there exists k', $\boldsymbol{m}$ such that $\Sigma; \bot_0 \vdash P \overset{\boldsymbol{m}}{\Longrightarrow} \Sigma' \vdash_{k'} P'$. The relation $\Rightarrow$ is well-founded for clock-guarded processes in the sense that it has no infinite chains.

Theorem 2 (Termination). *Let* $P_0, P_1, P_2, \ldots$ *and* $\Sigma_0, \Sigma_1, \Sigma_2, \ldots$ *be infinite sequences of processes and memories, respectively, with* $\Sigma_i \vdash P_i \Rightarrow \Sigma_{i+1} \vdash P_{i+1}$*. If* P_0 *is clock-guarded then there is* $n \geq 0$ *with* $\Sigma_n = \Sigma_i$*,* $P_n = P_i$ *for all* $i \geq n$*.*

The fixed point semantics will iterate (3) until it reaches a P^* such that $\Sigma^* \vdash P^* \Rightarrow \Sigma^* \vdash P^*$. By Termination Theorem 2 this must exist for clock-guarded processes. If $can^s(P^*) = \bot_0$ then P^* is 0-stable and the program P has terminated. If $can^s(P^*) = \bot_1$, the residual P^* is pausing.

Definition 6 (Macro Step). *A* run $\Sigma \vdash P \Rrightarrow \Sigma' \vdash P'$ *is a sequence of ssteps with processes* $P_0, P_1, P_2, \ldots, P_n$ *and sequences of method calls* $\boldsymbol{m}_1$*,* $\boldsymbol{m}_2, \ldots \boldsymbol{m}_n$ *such that for all* $1 \leq i \leq n$*,*

$$\Sigma_{i-1}; \bot_0 \vdash P_{i-1} \Rightarrow \Sigma_i \vdash_{k_i} P_i,$$

where $P_0 = P$*,* $\Sigma_0 = \Sigma$*,* $\Sigma_n = \Sigma'$ *and* $P_n = P'$*. A run is called a* macro-step *if it is maximal, i.e., if* $\Sigma' \vdash P' \Rightarrow \Sigma'' \vdash P''$ *implies* $\Sigma' = \Sigma''$ *and* $P' = P''$*. The macro-step is called* stabilising *if the final* P' *is stable, i.e.,* $k_n \neq \bot$ *and the clock is admissible, i.e., if* $(\Sigma'.\mathsf{c})^{\#} \odot \sigma$ *is defined for all* $\mathsf{c} \in \mathsf{O}$*. The macro-step is* pausing *if* $k_n = 1$ *and* terminating *if* $k_n = 0$*.* □

Given a pausing macro-step $\Sigma \vdash P \Rrightarrow \Sigma' \vdash P'$, then the next tick starts with process $\sigma(P')$ in memory Σ'' such that $(\pi_{\mathsf{c}}(\Sigma'))^{\#} \xrightarrow{\sigma} (\pi_{\mathsf{c}}(\Sigma''))^{\#}$ for all $\mathsf{c} \in \mathsf{O}$. This only constrains the abstract policy state of each variable in Σ'' not their memory content. In this way, CSM variables can introduce an arbitrary new memory Σ'' with every clock tick.

Theorem 3 (Macro-step Determinism). *If all* CSM *variables are policy-coherent then for two macro steps $\Sigma \vdash P \Rrightarrow \Sigma_1 \vdash P_1$ and $\Sigma \vdash P \Rrightarrow \Sigma_2 \vdash P_2$ we have $\Sigma_1 = \Sigma_2$ and $P_1 = P_2$.*

Definition 7 (Constructiveness). *A program P is* policy-constructive*, for a set of policy coherent* CSM *variables, if for arbitrary initial memory Σ all reachable macro-steps of P are stabilising.* □

A non-constructive program will, after some tick, end up in a fixed point P^* with $can^s(P^*) \not\subseteq \{\bot_0, \bot_1\}$. Then P^* is stuck involving a set of active child threads waiting for each other in a policy-induced cycle.

Finally, we present two important results for DCoL showing that we are conservatively extending existing SP semantics. A DCoL program using only sequentially constructive variables [14] (see [17] Sec. 5.7]) is called a *DCoL-SC* program. DCoL programs using only pure signals subject to the policy of Example 1 (Fig. 1) are expressive complete for the pure instantaneous fragment of Esterel [4]. Esterel signal emissions `emit s` are syntactic sugar for `s.emit();`. A presence test `pres s then` P `else` Q abbreviates `if s.pres() then` P `else` Q. Sequential composition P `;` Q in Esterel behaves like a parallel composition in which the schedule is forced to run P to termination before it can pass control to Q. In DCoL this is (P`;s'.emit();`) ‖ (`s'.pres() then` Q `else skip`) with fresh signal `s'` not occurring in either P or Q. This suggests the following definition: A program P is a *(pure instantaneous) DCoL-Esterel* program if (i) P only uses pure signals and (ii) P does not use `pause` or `rec` and (iii) P does not contain sequentially nested occurrences of signal accesses.

Theorem 4 (Esterel and Sequential Constructiveness)

1. *If an DCoL-Esterel program P is policy-constructive according to Definition 7 iff it is Berry-constructive in the sense of [4].*
2. *If a DCoL-SC program P is policy-constructive according to Definition 7 then it is sequentially constructive in the sense of [14].*

It is interesting to note that the second statement in Theorem 4 is not invertible (for a counter example see [17]). Hence, policy-constructiveness for SC-variables induced by our semantics is more restrictive than that given in [14].

4 Related Work

Many languages have been proposed to offer determinism as a fundamental design principle. We consider these attempts under several categories.

Fixed Protocol for Shared Data. These approaches introduce an unique protocol for data exchange between concurrent processes. SHIM [21] provides a model for combined hardware software systems typically of embedded systems. Here, the concurrent processes communicate using point-to-point (restricted) Kahn channels with blocking reads and writes. SHIM programs are shown to be deterministic-by-construction as the states of each process are finite and deterministic and the data produced-consumed over any channel is also deterministic.

Concurrent revisions [19] introduce a generic and deterministic programming model for parallel programming. This model supports fork-join parallelism and processes are allowed to make concurrent modifications to shared data by creating local copies that are eventually merged using suitable (programmer specified) merge functions at join boundaries.

However, like the deterministic SP model [2], both SHIM and concurrent revisions lack support for more expressive shared ADTs essential for programming complex systems. Caromel et al. [22], on the other hand, offer determinism with asynchronously communicating active objects (ADTs) equipped with a process calculus semantics. Here, an active object is a sequential thread. Active objects communicate using *futures* and synchronise via Kahn-MacQueen co-routines [23] for deterministic data exchange. Our approach subsumes Kahn buffers of SHIM and the *local-copy-merge protocol* of concurrent revisions by an appropriate choice of method interface and policy. None of these approaches [19,21,22] uses a clock as a central barrier mechanism like our approach does.

In the Java-derived language X10, clocks are a form of synchronisation barrier for supporting deterministic and deadlock-free patterns of common parallel computations [24]. This allows multiple-clocks in contrast to our approach. These, however, are not abstracted in the objects in contrast to our clocks that are encapsulated inside the CSM types. Hence X10 clocks are invoked directly by the *activities* (i.e., concurrent threads) of programs and this manual synchronisation is as error-prone as other unsafe low-level primitives such as locks.

Coherent Memory Models for Shared Data. Whether clocked or not, our approach depends on the availability of CSM types that are provably coherent for their policy. Besides the standard types of SP (data-flow, sequentially constructive variables, Kahn channels, signals) such CSM types can be obtained from existing research on *coherent memory models* [25,26]. Unlike the protocol-oriented approaches above, some approaches have been developed based on coherency of the underlying memory models [26] especially for shared objects.

Bocchino et al. [25] propose deterministic parallel Java (DPJ) which has a type and effect system to ensure that parallel heap accesses remain safe. Data structures such as arrays, trees, and sets can be accessed in parallel as long as accesses can be shown to use non-overlapping regions.

Grace [27] promises a deterministic run-time through the adoption of *fork-join* parallelism combined with memory protection and a sequential commit protocol. However, there is no guarantee on the determinism of such custom synchronisation protocols. These must be verified using expensive proof systems.

A powerful technique to generate coherent shared memory structure for functional programs has recently been proposed by Kuper et al. [28]. They introduce lattice-based data structures, called LVars, in which all write accesses produce a monotonic value increase in the lattice and all read accesses are blocked until the memory value has passed a read-specific threshold. Each variable's domain is organised as a lattice of states with $\perp$ and $\top$ representing an empty new location and an error, respectively. Because of monotonicity all writes are confluent with each other. Since reads are blocked each LVar data type can thus be used in DCoL as a coherent CSM type of variables with a threshold-determined policy. Note that [25–28] do not consider CSM types and [28] also do not treat destructive sequential updates as we do.

Recently Haller et al. [29] have developed Reactive Async, a new event-based asynchronous concurrent programming model that improves on LVars. This approach extends futures and promises[8] with lattice-based operations in order to support destructive updates (refinement of results) in a deterministic concurrent setting. The basic abstractions are: *cells* which define interfaces for reading a value that is asynchronously computed and (ii) *cell completers* that allow multiple monotonic updates of values taken from a lattice type class. The model supports concurrent programming with cyclic data dependencies in contrast to LVars. The mechanism for resolving cycles combines the lattices with quiescence detection on a handler pool (execution context). The quiescence concept refers to a state where the cell values are not going to be changed anymore. The thread pool is able to detect this quiescent (synchronisation) phase and when this is the case the resolution of cyclic dependencies and reading of cells can take place. This is similar to our policies, where enabling of methods (e. g., read) is a state and prediction-dependent notion. Our developments may offer a theoretical background for the cell interfaces of this model. In Reactive Async the concurrent code is guaranteed to be deterministic provided that the API is used appropriately but this is not checked statically. It would be interesting to investigate whether our theory can contribute on this front. In the other direction, Reactive Async manages inter-cell dependencies which might support global policies between different CSM variables in our setting.

Clock-Driven Encapsulation. Encapsulation is not entirely unknown in reactive programming. The idea of *reactive object model (ROM)* [30] was first introduced by Boussinot et al. and subsequently refined [31] and combined with standards such as UML [32]. Here a program is a collection of reactive objects that operate synchronously relative to a global clock, similar to SP. Each object encapsulates a set of methods and data, where the methods share this data. ROM relied on a simplified assumption, where each method invocation is separated into instants.

André et al. [33] generalised the ROM idea to that of *synchronous objects*, which behave like synchronous modules (in Esterel or Lustre). The program is divided into a collection of synchronous and standard objects. While the latter interact using messages, the former use *signals* like in SP. Communication between standard and synchronous objects has to be managed using special *interface objects*. The framework supports features such as aggregation, encapsulation and inheritance yet communication is restricted to standard Esterel-style signals. However, the issue of determinism for the composition of synchronous objects with standard objects is not considered.

[8] A future can asynchronously be completed with a value of the appropriate type or it can fail with an exception. A promise allows completing a future at most once.

A concrete implementation of synchronous objects in Java is proposed in [34]. Here, a run-time system is used to provide a cyclic schedule of the objects during an instant. This approach assumes that outputs from the objects can be read only in the next instant (similar to the SL programming language [35]) and so does not support instantaneous communication like we do.

Synchronous objects arise naturally in modular compilation [15,36,37]. The first time these have been exposed at the language level is in [20]. That work has inspired our use of policies. While [20] offers a mechanism for deterministic management of shared variables through ADT-like interfaces it has three serious limitations: (1) Modes express data-flow equations rather than imperative method procedures and so are not directly suitable for control-flow programming; (2) Policies do not distinguish between two modes being called *sequentially* by the *same* thread, which can be permitted, and two methods being called by *different* threads in *parallel*, which may have to be prohibited. This makes policies too restrictive in the light of the recent more liberal notion of sequential constructiveness [14] and, most importantly, (3) the notion of policy-soundness does not use policies *prescriptively* as a contract to be fulfilled by the scheduler but instead only *descriptively* as an invariant of the program code. Hence, policies in [20] cannot be used to generalise the semantics of SP signals to shared ADTs.

The sequentially constructive model of synchronous computation [14] has shown how the constructive semantics of Esterel can be reconstructed from a scheduling view as standard destructive variables plus synchronisation protocol. SCL acts as an intermediate language for the graphical language SCCharts [38] and the textual language SCEst [18] which are proposed as sequentially constructive extensions of the well-known control-flow languages SyncCharts [39] and Esterel [4]. By presenting our new analysis of sequential constructiveness for SCL our results become applicable both for SCCharts and SCEst.

The term ‘constructive’ semantics has been coined by Berry [4]. In [40] it was shown how it can be recoded as a fixed-point in an interval domain which we generalise here to policy states $[\mu, \gamma]$. Talpin et al. [13] recently gave a constructive semantics of multi-clock synchronous programs. It is an open problem how our approach could be generalised to multiple clocks.

5 Conclusion

This work extends the SP theoretical foundations to allow communication at higher levels of abstraction. The paper explains deterministic concurrency of SP as a derived property from CSM types. Our results extend the SP-notion of constructiveness to general shared CSM types. We have made some simplifying assumptions that render the theory somewhat less general than it could be. A first limitation is our assumption that all method calls are atomic. We believe the theory can be generalised for non-atomic methods albeit at the price of a significant increase in the complexity of calculating *can* predictions. Second, method parameters are passed “by value” rather than “by reference”. This is necessary for having types as black boxes ready to use. Method parameters passing variables "by reference" would also introduce aliasing issues which we do not address. Third, in our present setting the policy update $\mu \odot m$ does not observe method parameters. This is an abstraction to facilitate static analyses. In principle, to increase expressiveness, the method parameters could be included, too, but again complicate over-approximation for *can* information.

Acknowledgement. We thank Philipp Haller, Adrien Guatto and the three anonymous reviewers for their insightful comments and suggestions helping us improving the paper. This work has been supported by the German Research Council (DFG) under grant number ME-1427/6-2.

References

1. Lee, E.: The problem with threads. Computer **39**(5), 33–42 (2006)
2. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Guernic, P.L., de Simone, R.: The synchronous languages twelve years later. Proc. IEEE **91**(1), 64–83 (2003)
3. Colaço, J., Pagano, B., Pouzet, M.: SCADE 6: a formal language for embedded critical software development. In: TASE 2017, Sophia Antipolis, France, September 2017
4. Berry, G.: The Constructive Semantics of Pure Esterel. Draft Book (1999)
5. Schneider, K.: The synchronous programming language quartz. Internal report 375, Department of Computer Science, University of Kaiserslautern, Germany, December 2009
6. von Hanxleden, R.: SyncCharts in C – a proposal for light-weight, deterministic concurrency. In: EMSOFT 2009, Grenoble, France, pp. 225–234, October 2009
7. Guernic, P.L., Goutier, T., Borgne, M.L., Maire, C.L.: Programming real time applications with SIGNAL. Proc. IEEE **79**, 1321–1336 (1991)
8. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. Proc. IEEE **79**(9), 1305–1320 (1991)
9. Pouzet, M.: Lucid Synchrone, un langage synchrone d'ordre supérieur. Mémoire d'habilitation, Université Paris 6, November 2002
10. The Esterel v7 Reference Manual Version v7_30, November 2005
11. Aguado, J., Mendler, M.: Constructive semantics for instantaneous reactions. Theor. Comput. Sci. **412**, 931–961 (2011)
12. Aguado, J., Mendler, M., von Hanxleden, R., Fuhrmann, I.: Grounding synchronous deterministic concurrency in sequential programming. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 229–248. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_13
13. Talpin, J., Brandt, J., Gemünde, M., Schneider, K., Shukla, S.: Constructive polychronous systems. Sci. Comput. Prog. **96**(3), 377–394 (2014)
14. von Hanxleden, R., Mendler, M., Aguado, J., Duderstadt, B., Fuhrmann, I., Motika, C., Mercer, S., O'Brien, O., Roop, P.: Sequentially constructive concurrency—a conservative extension of the synchronous model of computation. ACM TECS **13**(4s), 144:1–144:26 (2014)
15. Pouzet, M., Raymond, P.: Modular static scheduling of synchronous data-flow networks - an efficient symbolic representation. Des. Autom. Embed. Syst. **14**(3), 165–192 (2010)

16. Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress 1974, Stockholm, Sweden, pp. 471–475, August 1974
17. Aguado, J., Mendler, M., Pouzet, M., Roop, P., von Hanxleden, R.: Clock-synchronised shared objects for deterministic concurrency. Research report 102, University of Bamberg, Germany, July 2017. https://www.uni-bamberg.de/fileadmin/uni/fakultaeten/wiai_professuren/grundlagen_informatik/papersMM/report-WIAI-102-Feb-2018.pdf
18. Rathlev, K., Smyth, S., Motika, C., von Hanxleden, R., Mendler, M.: SCEst: sequentially constructive esterel. ACM TECS **17**(2), 33:1–33:26 (2018)
19. Burckhardt, S., Leijen, D., Fähndrich, M., Sagiv, M.: Eventually consistent transactions. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 67–86. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_4
20. Caspi, P., Colaço, J., Gérard, L., Pouzet, M., Raymond, P.: Synchronous objects with scheduling policies: introducing safe shared memory in lustre. In: LCTES 2009, Dublin, Ireland, pp. 11–20, June 2009
21. Vasudevan, N.: Efficient, deterministic and deadlock-free concurrency. Ph.D. thesis, Department of Computer Science, Columbia University, March 2011
22. Caromel, D., Henrio, L., Serpette, B.: Asynchronous and deterministic objects. In: POPL 2004, Venice, Italy, pp. 123–134, January 2004
23. Kahn, G., MacQueen, D.: Coroutines and networks of parallel processes. In: IFIP Congress 1977, Toronto, Canada, pp. 993–998, August 1977
24. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA 2005, San Diego, USA, pp. 519–538, October 2005
25. Bocchino, R., Adve, V., Dig, D., Adve, S., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A type and effect system for deterministic parallel Java. In: OOPSLA 2009, Orlando, USA, pp. 97–116, October 2009
26. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI 2003, San Diego, USA, pp. 338–349, June 2003
27. Berger, E., Yang, T., Liu, T., Novark, G.: Grace: safe multithreaded programming for C/C++. In: OOPSLA 2009, Orlando, USA, pp. 81–96, October 2009
28. Kuper, L., Turon, A., Krishnaswami, N., Newton, R.: Freeze after writing: quasi-deterministic parallel programming with LVars. In: POPL 2014, San Diego, USA, pp. 257–270, January 2014
29. Haller, P., Geries, S., Eichberg, M., Salvaneschi, G.: Reactive Async: expressive deterministic concurrency. In: SCALA 2016, Amsterdam, Netherlands, pp. 11–20, October 2016
30. Boussinot, F., Doumenc, G., Stefani, J.: Reactive objects. Annales des télécommunications **51**(9–10), 459–473 (1996)
31. Talpin, J., Benveniste, A., Caillaud, B., Jard, C., Bouziane, Z., Canon, H.: BDL, a language of distributed reactive objects. In: IEEE ISORC 1998, Kyoto, Japan, pp. 196–205, April 1998
32. André, C., Peraldi-Frati, M.-A., Rigault, J.-P.: Integrating the synchronous paradigm into UML: application to control-dominated systems. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 163–178. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45800-X_15
33. André, C., Boulanger, F., Péraldi, M., Rigault, J., Vidal-Naquet, G.: Objects and synchronous programming. RAIRO-APII-JESA-J. Eur. Syst. Autom. **31**(3), 417–432 (1997)

34. Passerone, C., Sansoe, C., Lavagno, L., McGeer, R., Martin, J., Passerone, R., Sangiovanni-Vincentelli, A.: Modeling reactive systems in Java. ACM TODAES **3**(4), 515–523 (1998)
35. Boussinot, F., Simone, R.D.: The SL synchronous language. IEEE TSE **22**(4), 256–266 (1996)
36. Biernacki, D., Colaço, J., Hamon, G., Pouzet, M.: Clock-directed modular code generation of synchronous data-flow languages. In: LCTES 2008, Tucson, USA, pp. 121–130, June 2008
37. Hainque, O., Pautet, L., Le Biannic, Y., Nassor, É.: Cronos: a separate compilation tool set for modular Esterel applications. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1836–1853. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48118-4_47
38. von Hanxleden, R., Duderstadt, B., Motika, C., Smyth, S., Mendler, M., Aguado, J., Mercer, S., O'Brien, O.: SCCharts: sequentially constructive statecharts for safety-critical applications. SIGPLAN Not. **49**(6), 372–383 (2014)
39. André, C.: Semantics of SyncCharts. Technical report ISRN I3S/RR-2003-24-FR, I3S Laboratory, Sophia-Antipolis, France, April 2003
40. Aguado, J., Mendler, M., von Hanxleden, R., Fuhrmann, I.: Denotational fixed-point semantics for constructive scheduling of synchronous concurrency. Acta Informatica **52**(4), 393–442 (2015)

Probabilistic Programming

An Assertion-Based Program Logic for Probabilistic Programs

Gilles Barthe[1], Thomas Espitau[2], Marco Gaboardi[3], Benjamin Grégoire[4], Justin Hsu[5](✉), and Pierre-Yves Strub[6]

[1] IMDEA Software Institute, Madrid, Spain
[2] Université Paris 6, Paris, France
[3] University at Buffalo, SUNY, Buffalo, USA
[4] Inria Sophia Antipolis–Méditerranée, Nice, France
[5] University College London, London, UK
[6] École Polytechnique, Palaiseau, France

Abstract. We present Ellora, a sound and relatively complete assertion-based program logic, and demonstrate its expressivity by verifying several classical examples of randomized algorithms using an implementation in the EasyCrypt proof assistant. Ellora features new proof rules for loops and adversarial code, and supports richer assertions than existing program logics. We also show that Ellora allows convenient reasoning about complex probabilistic concepts by developing a new program logic for probabilistic independence and distribution law, and then smoothly embedding it into Ellora.

1 Introduction

The most mature systems for deductive verification of randomized algorithms are *expectation-based* techniques; seminal examples include PPDL [28] and pGCL [34]. These approaches reason about *expectations*, functions E from states to real numbers,[1] propagating them backwards through a program until they are transformed into a mathematical function of the input. Expectation-based systems are both theoretically elegant [16,23,24,35] and practically useful; implementations have verified numerous randomized algorithms [19,21]. However, properties involving multiple probabilities or expected values can be cumbersome to verify—each expectation must be analyzed separately.

An alternative approach envisioned by Ramshaw [37] is to work with predicates over distributions. A direct comparison with expectation-based techniques

This is the conference version of the paper.

[1] Treating a program as a function from input states s to output distributions $\mu(s)$, the expected value of E on $\mu(s)$ is an expectation.

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-3-319-89884-1_5) contains supplementary material, which is available to authorized users.

is difficult, as the approaches are quite different. In broad strokes, assertion-based systems can verify richer properties in one shot and have specifications that are arguably more intuitive, especially for reasoning about loops, while expectation-based approaches can transform expectations mechanically and can reason about non-determinism. However, the comparison is not very meaningful for an even simpler reason: existing assertion-based systems such as [8,18,38] are not as well developed as their expectation-based counterparts.

Restrictive Assertions. Existing probabilistic program logics do not support reasoning about expected values, only probabilities. As a result, many properties about average-case behavior are not even expressible.

Inconvenient Reasoning for Loops. The Hoare logic rule for deterministic loops does not directly generalize to probabilistic programs. Existing assertion-based systems either forbid loops, or impose complex semantic side conditions to control which assertions can be used as loop invariants. Such side conditions are restrictive and difficult to establish.

No Support for External or Adversarial Code. A strength of expectation-based techniques is reasoning about programs combining probabilities and *non-determinism*. In contrast, Morgan and McIver [30] argue that assertion-based techniques cannot support compositional reasoning for such a combination. For many applications, including cryptography, we would still like to reason about a commonly-encountered special case: programs using external or adversarial code. Many security properties in cryptography boil down to analyzing such programs, but existing program logics do not support adversarial code.

Few Concrete Implementations. There are by now several independent implementations of expectation-based techniques, capable of verifying interesting probabilistic programs. In contrast, there are only scattered implementations of probabilistic program logics.

These limitations raise two points. Compared to expectation-based approaches:

1. Can assertion-based approaches achieve similar expressivity?
2. Are there situations where assertion-based approaches are more suitable?

In this paper, we give positive evidence for both of these points.[2] Towards the first point, we give a new assertion-based logic Ellora for probabilistic programs, overcoming limitations in existing probabilistic program logics. Ellora supports a rich set of assertions that can express concepts like expected values and probabilistic independence, and novel proof rules for verifying loops and adversarial code. We prove that Ellora is sound and relatively complete.

Towards the second point, we evaluate Ellora in two ways. First, we define a new logic for proving probabilistic independence and distribution law

[2] Note that we do not give mathematically precise formulations of these points; as we are interested in the practical verification of probabilistic programs, a purely theoretical answer would not address our concerns.

properties—which are difficult to capture with expectation-based approaches—and then embed it into Ellora. This sub-logic is more narrowly focused than Ellora, but supports more concise reasoning for the target assertions. Our embedding demonstrates that the assertion-based approach can be flexibly integrated with intuitive, special-purpose reasoning principles. To further support this claim, we also provide an embedding of the Union Bound logic, a program logic for reasoning about accuracy bounds [4]. Then, we develop a full-featured implementation of Ellora in the EasyCrypt theorem prover and exercise the logic by mechanically verifying a series of complex randomized algorithms. Our results suggest that the assertion-based approach can indeed be practically viable.

Abstract Logic. To ease the presentation, we present Ellora in two stages. First, we consider an abstract version of the logic where assertions are general predicates over distributions, with no compact syntax. Our abstract logic makes two contributions: reasoning for loops, and for adversarial code.

Reasoning About Loops. Proving a property of a probabilistic loop typically requires establishing a loop invariant, but the class of loop invariants that can be soundly used depends on the termination behavior—stronger termination assumptions allows richer loop invariants. We identify three classes of assertions that can be used for reasoning about probabilistic loops, and provide a proof rule for each one:

– arbitrary assertions for *certainly terminating* loops, i.e. loops that terminate in a finite amount of iterations;
– *topologically closed* assertions for *almost surely* terminating loops, i.e. loops terminating with probability 1;
– *downwards closed* assertions for arbitrary loops.

The definition of topologically closed assertion is reminiscent of Ramshaw [37]; the stronger notion of downwards closed assertion appears to be new.

Besides broadening the class of loops that can be analyzed, our rules often enable simpler proofs. For instance, if the loop is certainly terminating, then there is no need to prove semantic side-conditions. Likewise, there is no need to consider the termination behavior of the loop when the invariant is downwards and topologically closed. For example, in many applications in cryptography, the target property is that a "bad" event has low probability: $\Pr[E] \leq k$. In our framework this assertion is downwards and topologically closed, so it can be a loop invariant regardless of the termination behavior.

Reasoning About Adversaries. Existing assertion-based logics cannot reason about probabilistic programs with *adversarial* code. *Adversaries* are special probabilistic procedures consisting of an interface listing the concrete procedures that an adversary can call (*oracles*), along with restrictions like how many calls an adversary may make. Adversaries are useful in cryptography, where security notions are described using experiments in which adversaries interact with a challenger, and in game theory and mechanism design, where adversaries can represent strategic agents. Adversaries can also model inputs to *online* algorithms.

We provide proof rules for reasoning about adversary calls. Our rules are significantly more general than previously considered rules for reasoning about adversaries. For instance, the rule for adversary used by [4] is restricted to adversaries that cannot make oracle calls.

Metatheory. We show soundness and relative completeness of the core abstract logic, with mechanized proofs in the Coq proof assistant.

Concrete Logic. While the abstract logic is conceptually clean, it is inconvenient for practical formal verification—the assertions are too general and the rules involve semantic side-conditions. To address these issues, we flesh out a concrete version of Ellora. Assertions are described by a grammar modeling a two-level assertion language. The first level contains state predicates—deterministic assertions about a single memory—while the second layer contains probabilistic predicates constructed from probabilities and expected values over discrete distributions. While the concrete assertions are theoretically less expressive than their counterparts in the abstract logic, they can already encode common properties and notions from existing proofs, like probabilities, expected values, distribution laws and probabilistic independence. Our assertions can express theorems from probability theory, enabling sophisticated reasoning about probabilistic concepts.

Furthermore, we leverage the concrete syntax to simplify verification.

- We develop an automated procedure for generating pre-conditions of non-looping commands, inspired by expectation-based systems.
- We give syntactic conditions for the closedness and termination properties required for soundness of the loop rules.

Implementation and Case Studies. We implement Ellora on top of EasyCrypt, a general-purpose proof assistant for reasoning about probabilistic programs, and we mechanically verify a diverse collection of examples including textbook algorithms and a randomized routing procedure. We develop an EasyCrypt formalization of probability theory from the ground up, including tools like concentration bounds (e.g., the Chernoff bound), Markov's inequality, and theorems about probabilistic independence.

Embeddings. We propose a simple program logic for proving *probabilistic independence.* This logic is designed to reason about independence in a lightweight way, as is common in paper proofs. We prove that the logic can be embedded into Ellora, and is therefore sound. Furthermore, we prove an embedding of the Union Bound logic [4].

2 Mathematical Preliminaries

As is standard, we will model randomized computations using *sub-distributions.*

Definition 1. *A* sub-distribution *over a set A is defined by a mass function $\mu : A \to [0,1]$ that gives the probability of the unitary events $a \in A$. This mass function must be s.t. $\sum_{a \in A} \mu(a)$ is well-defined and $|\mu| \triangleq \sum_{a \in A} \mu(a) \leq 1$. In particular, the* support $\mathrm{supp}(\mu) \triangleq \{a \in A \mid \mu(a) \neq 0\}$ *is discrete.*[3] *The name "sub-distribution" emphasizes that the total probability may be strictly less than* 1*. When the* weight $|\mu|$ *is equal to* 1*, we call μ a* distribution*. We let* $\mathbf{SDist}(A)$ *denote the set of sub-distributions over A. The probability of an event $E(x)$ w.r.t. a sub-distribution μ, written* $\Pr_{x \sim \mu}[E(x)]$*, is defined as* $\sum_{x \in A \mid E(x)} \mu(x)$.

Simple examples of sub-distributions include the *null sub-distribution* $\mathbf{0}$, which maps each element of the underlying space to 0; and the *Dirac distribution centered on x*, written δ_x, which maps x to 1 and all other elements to 0. The following standard construction gives a monadic structure to sub-distributions.

Definition 2. *Let $\mu \in \mathbf{SDist}(A)$ and $f : A \to \mathbf{SDist}(B)$. Then $\mathbb{E}_{a \sim \mu}[f] \in \mathbf{SDist}(B)$ is defined by*

$$\mathbb{E}_{a \sim \mu}[f](b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b).$$

We use notation reminiscent of expected values, as the definition is quite similar.

We will need two constructions to model branching statements.

Definition 3. *Let $\mu_1, \mu_2 \in \mathbf{SDist}(A)$ such that $|\mu_1| + |\mu_2| \leq 1$. Then $\mu_1 + \mu_2$ is the sub-distribution μ such that $\mu(a) = \mu_1(a) + \mu_2(a)$ for every $a \in A$.*

Definition 4. *Let $E \subseteq A$ and $\mu \in \mathbf{SDist}(A)$. Then the restriction $\mu_{|E}$ of μ to E is the sub-distribution such that $\mu_{|E}(a) = \mu(a)$ if $a \in E$ and 0 otherwise.*

Sub-distributions are partially ordered under the pointwise order.

Definition 5. *Let $\mu_1, \mu_2 \in \mathbf{SDist}(A)$. We say $\mu_1 \leq \mu_2$ if $\mu_1(a) \leq \mu_2(a)$ for every $a \in A$, and we say $\mu_1 = \mu_2$ if $\mu_1(a) = \mu_2(a)$ for every $a \in A$.*

We use the following lemma when reasoning about the semantics of loops.

Lemma 1. *If $\mu_1 \leq \mu_2$ and $|\mu_1| = 1$, then $\mu_1 = \mu_2$ and $|\mu_2| = 1$.*

Sub-distributions are stable under pointwise-limits.

[3] We work with discrete distributions to keep measure-theoretic technicalities to a minimum, though we do not see obstacles to generalizing to the continuous setting.

Definition 6. *A sequence* $(\mu_n)_{n\in\mathbb{N}} \in \mathbf{SDist}(A)$ *sub-distributions* converges *if for every* $a \in A$*, the sequence* $(\mu_n(a))_{n\in\mathbb{N}}$ *of real numbers converges. The* limit sub-distribution *is defined as*

$$\mu_\infty(a) \triangleq \lim_{n\to\infty} \mu_n(a)$$

for every $a \in A$*. We write* $\lim_{n\to\infty} \mu_n$ *for* μ_∞*.*

Lemma 2. *Let* $(\mu_n)_{n\in\mathbb{N}}$ *be a convergent sequence of sub-distributions. Then for any event* $E(x)$*, we have:*

$$\forall n \in \mathbb{N}. \Pr_{x\sim\mu_\infty}[E(x)] = \lim_{n\to\infty} \Pr_{x\sim\mu_n}[E(x)].$$

Any bounded increasing real sequence has a limit; the same is true of sub-distributions.

Lemma 3. *Let* $(\mu_n)_{n\in\mathbb{N}} \in \mathbf{SDist}(A)$ *be an increasing sequence of sub-distributions. Then, this sequence converges to* μ_∞ *and* $\mu_n \leq \mu_\infty$ *for every* $n \in \mathbb{N}$*. In particular, for any event* E*, we have* $\Pr_{x\sim\mu_n}[E] \leq \Pr_{x\sim\mu_\infty}[E]$ *for every* $n \in \mathbb{N}$*.*

3 Programs and Assertions

Now, we introduce our core programming language and its denotational semantics.

Programs. We base our development on pWhile, a strongly-typed imperative language with deterministic assignments, probabilistic assignments, conditionals, loops, and an **abort** statement which halts the computation with no result. Probabilistic assignments $x \xleftarrow{\$} g$ assign a value sampled from a distribution g to a program variable x. The syntax of statements is defined by the grammar:

$$\begin{aligned} s ::= {} & \mathbf{skip} \mid \mathbf{abort} \mid x \leftarrow e \mid x \xleftarrow{\$} g \mid s; s \\ & \mid \mathbf{if}\ e\ \mathbf{then}\ s\ \mathbf{else}\ s \mid \mathbf{while}\ e\ \mathbf{do}\ s \mid x \leftarrow \mathcal{I}(e) \mid x \leftarrow \mathcal{A}(e) \end{aligned}$$

where x, e, and g range over typed variables in $\mathcal{X}$, expressions in $\mathcal{E}$ and distribution expressions in $\mathcal{D}$ respectively. The set $\mathcal{E}$ of well-typed expressions is defined inductively from $\mathcal{X}$ and a set $\mathcal{F}$ of function symbols, while the set $\mathcal{D}$ of well-typed distribution expressions is defined by combining a set of distribution symbols $\mathcal{S}$ with expressions in $\mathcal{E}$. Programs may call a set $\mathcal{I}$ of internal procedures as well as a set $\mathcal{A}$ of external procedures. We assume that we have code for internal procedures but not for external procedures—we only know indirect information, like which internal procedures they may call. Borrowing a convention from cryptography, we call internal procedures *oracles* and external procedures *adversaries.*

Semantics. The denotational semantics of programs is adapted from the seminal work of [27] and interprets programs as sub-distribution transformers. We view

$$
\begin{aligned}
[\![\mathbf{skip}]\!]_m &= \delta_m \\
[\![\mathbf{abort}]\!]_m &= \mathbf{0} \\
[\![x \leftarrow e]\!]_m &= \delta_{m[x:=[\![e]\!]_m]} \\
[\![x \overset{\$}{\leftarrow} g]\!]_m &= \mathbb{E}_{v\sim[\![g]\!]_m}[\delta_{m[x:=v]}] \\
[\![s_1; s_2]\!]_m &= \mathbb{E}_{m'\sim[\![s_1]\!]_m}[[\![s_2]\!]_{m'}] \\
[\![\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\!]_m &= \text{if } [\![e]\!]_m \text{ then } [\![s_1]\!]_m \text{ else } [\![s_2]\!]_m \\
[\![\mathbf{while}\ e\ \mathbf{do}\ s]\!]_m &= \lim_{n\to\infty} [\![(\mathbf{if}\ e\ \mathbf{then}\ s)^n; \mathbf{if}\ e\ \mathbf{then}\ \mathbf{abort}]\!]_m \\
[\![x \leftarrow \mathcal{I}(e)]\!]_m &= [\![f_{\mathbf{arg}} \leftarrow e; f_{\mathbf{body}}; x \leftarrow f_{\mathbf{res}}]\!]_m \\
[\![x \leftarrow \mathcal{A}(e)]\!]_m &= [\![a_{\mathbf{arg}} \leftarrow e; a_{\mathbf{body}}; x \leftarrow a_{\mathbf{res}}]\!]_m
\end{aligned}
$$

$$[\![s]\!]_\mu = \mathbb{E}_{m\sim\mu}[[\![s]\!]_m]$$

Fig. 1. Denotational semantics of programs

states as type-preserving mappings from variables to values; we write **State** for the set of states and **SDist**(**State**) for the set of probabilistic states. For each procedure name $f \in \mathcal{I} \cup \mathcal{A}$, we assume a set $\mathcal{X}_f^{\mathfrak{L}} \subseteq \mathcal{X}$ of *local variables* s.t. $\mathcal{X}_f^{\mathfrak{L}}$ are pairwise disjoint. The other variables $\mathcal{X} \setminus \bigcup_f \mathcal{X}_f^{\mathfrak{L}}$ are *global variables*.

To define the interpretation of expressions and distribution expressions, we let $[\![e]\!]_m$ denote the interpretation of expression e with respect to state m, and $[\![e]\!]_\mu$ denote the interpretation of expression e with respect to an initial sub-distribution μ over states defined by the clause $[\![e]\!]_\mu \triangleq \mathbb{E}_{m\sim\mu}[[\![e]\!]_m]$. Likewise, we define the semantics of commands in two stages: first interpreted in a single input memory, then interpreted in an input sub-distribution over memories.

Definition 7. *The semantics of commands are given in Fig. 1.*

- *The semantics $[\![s]\!]_m$ of a statement s in initial state m is a sub-distribution over states.*
- *The (lifted) semantics $[\![s]\!]_\mu$ of a statement s in initial sub-distribution μ over states is a sub-distribution over states.*

We briefly comment on loops. The semantics of a loop **while** e **do** c is defined as the limit of its lower approximations, where the n-th *lower approximation* of $[\![\mathbf{while}\ e\ \mathbf{do}\ c]\!]_\mu$ is $[\![(\mathbf{if}\ e\ \mathbf{then}\ s)^n; \mathbf{if}\ e\ \mathbf{then}\ \mathbf{abort}]\!]_\mu$, where **if** e **then** s is shorthand for **if** e **then** s **else skip** and c^n is the n-fold composition $c; \cdots; c$. Since the sequence is increasing, the limit is well-defined by Lemma 3. In contrast, the n-th *approximation* of $[\![\mathbf{while}\ e\ \mathbf{do}\ c]\!]_\mu$ defined by $[\![(\mathbf{if}\ e\ \mathbf{then}\ s)^n]\!]_\mu$ may not converge, since they are not necessarily increasing. However, in the special case where the output distribution has weight 1, the n-th lower approximations and the n-th approximations have the same limit.

Lemma 4. *If the sub-distribution* $[\![\textbf{while } e \textbf{ do } c]\!]_\mu$ *has weight* 1*, then the limit of* $[\![(\textbf{if } e \textbf{ then } s)^n]\!]_\mu$ *is defined and*

$$\lim_{n\to\infty} [\![(\textbf{if } e \textbf{ then } s)^n; \textbf{if } e \textbf{ then abort}]\!]_\mu = \lim_{n\to\infty} [\![(\textbf{if } e \textbf{ then } s)^n]\!]_\mu.$$

This follows by Lemma 1, since lower approximations are below approximations so the limit of their weights (and the weight of their limit) is 1. It will be useful to identify programs that terminate with probability 1.

Definition 8 (Lossless). *A statement s is* lossless *if for every sub-distribution μ, $|[\![s]\!]_\mu| = |\mu|$, where $|\mu|$ is the total probability of μ. Programs that are not lossless are called* lossy.

Informally, a program is lossless if all probabilistic assignments sample from full distributions rather than sub-distributions, there are no **abort** instructions, and the program is almost surely terminating, i.e. infinite traces have probability zero. Note that if we restrict the language to sample from full distributions, then losslessness coincides with almost sure termination.

Another important class of loops are loops with a uniform upper bound on the number of iterations. Formally, we say that a loop **while** e **do** s is *certainly terminating* if there exists k such that for every sub-distribution μ, we have $|[\![\textbf{while } e \textbf{ do } s]\!]_\mu| = |[\![(\textbf{if } e \textbf{ then } s)^k]\!]_\mu|$. Note that certain termination of a loop does not entail losslessness—the output distribution of the loop may not have weight 1, for instance, if the loop samples from a sub-distribution or if the loop aborts with positive probability.

Semantics of Procedure Calls and Adversaries. The semantics of internal procedure calls is straightforward. Associated to each procedure name $f \in \mathcal{I}$, we assume a designated input variable $f_{\textbf{arg}} \in \mathcal{X}_f^{\mathfrak{L}}$, a piece of code $f_{\textbf{body}}$ that executes the function call, and a result expression $f_{\textbf{res}}$. A function call $x \leftarrow \mathcal{I}(e)$ is then equivalent to $f_{\textbf{arg}} \leftarrow e; f_{\textbf{body}}; x \leftarrow f_{\textbf{res}}$. Procedures are subject to well-formedness criteria: procedures should only use local variables in their scope and after initializing them, and should not perform recursive calls.

External procedure calls, also known as adversary calls, are a bit more involved. Each name $a \in \mathcal{A}$ is parametrized by a set $a_{\textbf{ocl}} \subseteq \mathcal{I}$ of internal procedures which the adversary may call, a designated input variable $a_{\textbf{arg}} \in \mathcal{X}_a^{\mathfrak{L}}$, a (unspecified) piece of code $a_{\textbf{body}}$ that executes the function call, and a result expression $a_{\textbf{res}}$. We assume that adversarial code can only access its local variables in $\mathcal{X}_a^{\mathfrak{L}}$ and can only make calls to procedures in $a_{\textbf{ocl}}$. It is possible to impose more restrictions on adversaries—say, that they are lossless—but for simplicity we do not impose additional assumptions on adversaries here.

4 Proof System

In this section we introduce a program logic for proving properties of probabilistic programs. The logic is abstract—assertions are arbitrary predicates on sub-distributions—but the meta-theoretic properties are clearest in this setting. In the following section, we will give a concrete version suitable for practical use.

Assertions and Closedness Conditions. We use predicates on state distribution.

Definition 9 (Assertions). *The set* Assn *of assertions is defined as* $\mathcal{P}(\mathbf{SDist}(\mathbf{State}))$*. We write* $\eta(\mu)$ *for* $\mu \in \eta$*.*

Usual set operations are lifted to assertions using their logical counterparts, e.g., $\eta \wedge \eta' \triangleq \eta \cap \eta'$ and $\neg\eta \triangleq \overline{\eta}$. Our program logic uses a few additional constructions. Given a predicate ϕ over states, we define

$$\Box\phi(\mu) \triangleq \forall m.\, m \in \mathrm{supp}(\mu) \implies \phi(m)$$

where $\mathrm{supp}(\mu)$ is the set of all states with non-zero probability under μ. Intuitively, ϕ holds deterministically on all states that we may sample from the distribution. To reason about branching commands, given two assertions η_1 and η_2, we let

$$(\eta_1 \oplus \eta_2)(\mu) \triangleq \exists \mu_1, \mu_2 \;.\; \mu = \mu_1 + \mu_2 \wedge \eta_1(\mu_1) \wedge \eta_2(\mu_2).$$

This assertion means that the sub-distribution is the sum of two sub-distributions such that η_1 holds on the first piece and η_2 holds on the second piece.

Given an assertion η and an event $E \subseteq \mathbf{State}$, we let $\eta_{|E}(\mu) \triangleq \eta(\mu_{|E})$. This assertion holds exactly when η is true on the portion of the sub-distribution satisfying E. Finally, given an assertion η and a function F from $\mathbf{SDist}(\mathbf{State})$ to $\mathbf{SDist}(\mathbf{State})$, we define $\eta[F] \triangleq \lambda\mu.\, \eta(F(\mu))$. Intuitively, $\eta[F]$ is true in a sub-distribution μ exactly when η holds on $F(\mu)$.

Now, we can define the closedness properties of assertions. These properties will be critical to our rules for **while** loops.

Definition 10 (Closedness properties). *A family of assertions* $(\eta_n)_{n \in \mathbb{N}^\infty}$ *is:*

- u-closed *if for every increasing sequence of sub-distributions* $(\mu_n)_{n\in\mathbb{N}}$ *such that* $\eta_n(\mu_n)$ *for all* $n \in \mathbb{N}$ *then* $\eta_\infty(\lim_{n\to\infty} \mu_n)$*;*
- t-closed *if for every converging sequence of sub-distributions* $(\mu_n)_{n\in\mathbb{N}}$ *such that* $\eta_n(\mu_n)$ *for all* $n \in \mathbb{N}$ *then* $\eta_\infty(\lim_{n\to\infty} \mu_n)$*;*
- d-closed *if it is* t*-closed and downward closed, that is for every sub-distributions* $\mu \le \mu'$, $\eta_\infty(\mu')$ *implies* $\eta_\infty(\mu)$*.*

When $(\eta_n)_n$ *is constant and equal to* η*, we say that* η *is u-/t-/d-closed.*

Note that t-closedness implies u-closedness, but the converse does not hold. Moreover, u-closed, t-closed and d-closed assertions are closed under arbitrary intersections and finite unions, or in logical terms under finite boolean combinations, universal quantification over arbitrary sets and existential quantification over finite sets.

Finally, we introduce the necessary machinery for the frame rule. The set $\mathrm{mod}(s)$ of *modified* variables of a statement s consists of all the variables on the left of a deterministic or probabilistic assignment. In this setting, we say that an assertion η is *separated* from a set of variables X, written $\mathsf{separated}(\eta, X)$, if $\eta(\mu_1) \iff \eta(\mu_2)$ for any distributions μ_1, μ_2 s.t. $|\mu_1| = |\mu_2|$ and $\mu_{1|\overline{X}} = \mu_{2|\overline{X}}$ where for a set of variables X, the restricted sub-distribution $\mu_{|X}$ is

$$\mu_{|X} : m \in \mathbf{State}_{|X} \mapsto \Pr_{m' \sim \mu}[m = m'_{|X}]$$

where $\mathbf{State}_{|X}$ and $m_{|X}$ restrict $\mathbf{State}$ and m to the variables in X.

Intuitively, an assertion is separated from a set of variables X if every two sub-distributions that agree on the variables outside X either both satisfy the assertion, or both refute the assertion.

Judgments and Proof Rules. Judgments are of the form $\{\eta\}\, s\, \{\eta'\}$, where the assertions η and η' are drawn from Assn.

Definition 11. *A judgment* $\{\eta\}\, s\, \{\eta'\}$ *is* valid*, written* $\models \{\eta\}\, s\, \{\eta'\}$*, if* $\eta'(\llbracket s \rrbracket_\mu)$ *for every interpretation of adversarial procedures and every probabilistic state* μ *such that* $\eta(\mu)$*.*

Figure 2 describes the structural and basic rules of the proof system. Validity of judgments is preserved under standard structural rules, like the rule of consequence [Conseq]. As usual, the rule of consequence allows to weaken the post-condition and to strengthen the post-condition; in our system, this rule serves as the interface between the program logic and mathematical theorems from probability theory. The [Exists] rule is helpful to deal with existentially quantified pre-conditions.

$$\frac{\eta_0 \Rightarrow \eta_1 \qquad \{\eta_1\}\ s\ \{\eta_2\} \qquad \eta_2 \Rightarrow \eta_3}{\{\eta_0\}\ s\ \{\eta_3\}}\ [\textsc{Conseq}] \qquad \frac{\forall x : T.\, \{\eta\}\ s\ \{\eta'\}}{\{\exists x : T.\, \eta\}\ s\ \{\eta'\}}\ [\textsc{Exists}]$$

$$\frac{}{\{\eta\}\ \mathbf{abort}\ \{\Box\bot\}}\ [\textsc{Abort}] \qquad \frac{\eta' \triangleq \eta[\llbracket x \leftarrow e \rrbracket]}{\{\eta'\}\ x \leftarrow e\ \{\eta\}}\ [\textsc{Assgn}] \qquad \frac{}{\{\eta\}\ \mathbf{skip}\ \{\eta\}}\ [\textsc{Skip}]$$

$$\frac{\eta' \triangleq \eta[\llbracket x \overset{\$}{\leftarrow} g \rrbracket]}{\{\eta'\}\ x \overset{\$}{\leftarrow} g\ \{\eta\}}\ [\textsc{Sample}] \qquad \frac{\{\eta_0\}\ s_1\ \{\eta_1\} \qquad \{\eta_1\}\ s_2\ \{\eta_2\}}{\{\eta_0\}\ s_1; s_2\ \{\eta_2\}}\ [\textsc{Seq}]$$

$$\frac{\{\eta_1 \land \Box e\}\ s_1\ \{\eta_1'\} \qquad \{\eta_2 \land \Box\neg e\}\ s_2\ \{\eta_2'\}}{\{(\eta_1 \land \Box e) \oplus (\eta_2 \land \Box\neg e)\}\ \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \{\eta_1' \oplus \eta_2'\}}\ [\textsc{Cond}]$$

$$\frac{\{\eta_1\}\ s\ \{\eta_1'\} \qquad \{\eta_2\}\ s\ \{\eta_2'\}}{\{\eta_1 \oplus \eta_2\}\ s\ \{\eta_1' \oplus \eta_2'\}}\ [\textsc{Split}]$$

$$\frac{\mathsf{separated}(\eta, \mathrm{mod}(s)) \qquad s \text{ is lossless}}{\{\eta\}\ s\ \{\eta\}}\ [\textsc{Frame}]$$

$$\frac{\{\eta\}\ f_{\mathbf{arg}} \leftarrow e; f_{\mathbf{body}}\ \{\eta'[\llbracket x \leftarrow f_{\mathbf{res}} \rrbracket]\}}{\{\eta\}\ x \leftarrow f(e)\ \{\eta'\}}\ [\textsc{Call}]$$

Fig. 2. Structural and basic rules

The rules for **skip**, assignments, random samplings and sequences are all straightforward. The rule for **abort** requires $\square\bot$ to hold after execution; this assertion uniquely characterizes the resulting null sub-distribution. The rules for assignments and random samplings are semantical.

The rule [Cond] for conditionals requires that the post-condition must be of the form $\eta_1 \oplus \eta_2$; this reflects the semantics of conditionals, which splits the initial probabilistic state depending on the guard, runs both branches, and recombines the resulting two probabilistic states.

The next two rules ([Split] and [Frame]) are useful for local reasoning. The [Split] rule reflects the additivity of the semantics and combines the pre- and post-conditions using the $\oplus$ operator. The [Frame] rule asserts that lossless statements preserve assertions that are not influenced by modified variables.

The rule [Call] for internal procedures is as expected, replacing the procedure call f with its definition.

Figure 3 presents the rules for loops. We consider four rules specialized to the termination behavior. The [While] rule is the most general rule, as it deals with arbitrary loops. For simplicity, we explain the rule in the special case where the family of assertions is constant, i.e. we have $\eta_n = \eta$ and $\eta'_n = \eta'$. Informally, the η is the loop invariant and η' is an auxiliary assertion used to prove the invariant. We require that η is u-closed, since the semantics of a loop is defined as the limit of its lower approximations. Moreover, the first premise ensures that starting from η, one guarded iteration of the loop establishes η'; the second premise ensures that restricting to $\neg e$ a probabilistic state μ' satisfying η' yields a probabilistic state μ satisfying η. It is possible to give an alternative formulation where the second premise is substituted by the logical constraint $\eta'_{|\neg e} \implies \eta$. As usual, the post-condition of the loop is the conjunction of the invariant with the negation of the guard (more precisely in our setting, that the guard has probability 0).

The [While-AST] rule deals with lossless loops. For simplicity, we explain the rule in the special case where the family of assertions is constant, i.e. we have $\eta_n = \eta$. In this case, we know that lower approximations and approximations have the same limit, so we can directly prove an invariant that holds after one guarded iteration of the loop. On the other hand, we must now require that the η satisfies the stronger property of t-closedness.

The [While-D] rule handles arbitrary loops with a d-closed invariant; intuitively, restricting a sub-distribution that satisfies a downwards closed assertion η yields a sub-distribution which also satisfies η.

The [While-CT] rule deals with certainly terminating loops. In this case, there is no requirement on the assertions.

We briefly compare the rules from a verification perspective. If the assertion is d-closed, then the rule [While-D] is easier to use, since there is no need to prove any termination requirement. Alternatively, if we can prove certain termination of the loop, then the rule [While-CT] is the best to use since it does not impose any condition on assertions. When the loop is lossless, there is no need to introduce an auxiliary assertion η', which simplifies the proof goal.

$$\frac{\begin{array}{c}\mathsf{uclosed}((\eta'_n)_{n\in\mathbb{N}^\infty}) \\ \forall n.\,\{\eta_n\}\ \textbf{if}\ e\ \textbf{then}\ s\ \{\eta_{n+1}\} \qquad \forall n.\,\{\eta_n\}\ \textbf{if}\ e\ \textbf{then}\ \textbf{abort}\ \{\eta'_n\}\end{array}}{\{\eta_0\}\ \textbf{while}\ e\ \textbf{do}\ s\ \{\eta'_\infty \wedge \Box\neg e\}}\ [\textsc{While}]$$

$$\frac{\begin{array}{c}\mathsf{tclosed}((\eta_n)_{n\in\mathbb{N}^\infty}) \qquad \forall n.\,\{\eta_n\}\ \textbf{if}\ e\ \textbf{then}\ s\ \{\eta_{n+1}\} \\ \forall \mu.\,\eta_0(\mu) \implies |[\![(\textbf{while}\ e\ \textbf{do}\ s)]\!]_\mu| = 1\end{array}}{\{\eta_0\}\ \textbf{while}\ e\ \textbf{do}\ s\ \{\eta_\infty \wedge \Box\neg e\}}\ [\textsc{While-AST}]$$

$$\frac{\mathsf{dclosed}((\eta_n)_{n\in\mathbb{N}^\infty}) \qquad \forall n.\,\{\eta_n\}\ \textbf{if}\ e\ \textbf{then}\ s\ \{\eta_{n+1}\}}{\{\eta_0\}\ \textbf{while}\ e\ \textbf{do}\ s\ \{\eta_\infty \wedge \Box\neg e\}}\ [\textsc{While-D}]$$

$$\frac{\begin{array}{c}\forall n.\,\{\eta_n\}\ \textbf{if}\ e\ \textbf{then}\ s\ \{\eta_{n+1}\} \\ \forall \mu.\,\eta_0(\mu) \implies [\![(\textbf{if}\ e\ \textbf{then}\ s)^k]\!]_\mu = [\![(\textbf{while}\ e\ \textbf{do}\ s)]\!]_\mu\end{array}}{\{\eta_0\}\ \textbf{while}\ e\ \textbf{do}\ s\ \{\eta_k \wedge \Box\neg e\}}\ [\textsc{While-CT}]$$

Fig. 3. Rules for loops

$$\frac{\begin{array}{c}\forall n \in \mathbb{N}^\infty.\ \mathsf{separated}(\eta_n, \{x, \mathfrak{s}\}) \qquad \mathsf{dclosed}((\eta_n)_{n\in\mathbb{N}^\infty}) \\ \forall f \in a_{\mathbf{ocl}}, x \in \mathcal{X}_a^{\mathfrak{L}}, e \in \mathcal{E}, n \in \mathbb{N}.\,\{\eta_n\}\ x \leftarrow f(e)\ \{\eta_{n+1}\}\end{array}}{\{\eta_0\}\ x \leftarrow a(e)\ \{\eta_\infty\}}\ [\textsc{Adv}]$$

Fig. 4. Rules for adversaries

Note however that it might still be beneficial to use the [While] rule, even for lossless loops, because of the weaker requirement that the invariant is u-closed rather than t-closed.

Finally, Fig. 4 gives the adversary rule for general adversaries. It is highly similar to the general rule [While-D] for loops since the adversary may make an arbitrary sequence of calls to the oracles in $a_{\mathbf{ocl}}$ and may not be lossless. Intuitively, η plays the role of the invariant: it must be d-closed and it must be preserved by every oracle call with arbitrary arguments. If this holds, then η is also preserved by the adversary call. Some framing conditions are required, similar to the ones of the [Frame] rule: the invariant must not be influenced by the state writable by the external procedures.

It is possible to give other variants of the adversary rule with more general invariants by restricting the adversary, e.g., requiring losslessness or bounding the number of calls the external procedure can make to oracles, leading to rules akin to the almost surely terminating and certainly terminating loop rules, respectively.

Soundness and Relative Completeness. Our proof system is sound with respect to the semantics.

Theorem 1 (Soundness). *Every judgment $\{\eta\}\, s\, \{\eta'\}$ provable using the rules of our logic is valid.*

Completeness of the logic follows from the next lemma, whose proof makes an essential use of the [While] rule. In the sequel, we use $\mathbf{1}_\mu$ to denote the characteristic function of a probabilistic state μ, an assertion stating that the current state is equal to μ.

Lemma 5. *For every probabilistic state μ, the following judgment is provable using the rule of the logic:*

$$\{\mathbf{1}_\mu\}\, s\, \{\mathbf{1}_{[\![s]\!]_\mu}\}.$$

Proof. By induction on the structure of s.

- $s = \mathbf{abort}$, $s = \mathbf{skip}$, $x \leftarrow e$ and $s = x \xleftarrow{\$} g$ are trivial;
- $s = s_1; s_2$, we have to prove

  $$\{\mathbf{1}_\mu\}\, s_1; s_2\, \{\mathbf{1}_{[\![s_2]\!]_{[\![s_1]\!]_\mu}}\}.$$

  We apply the [Seq] rule with $\eta_1 = \mathbf{1}_{[\![s_1]\!]_\mu}$ premises can be directly proved using the induction hypothesis;
- $s = \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2$, we have to prove

  $$\{\mathbf{1}_\mu\}\, \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\, \{(\mathbf{1}_{[\![s_1]\!]_{\mu_{|e}}} \oplus \mathbf{1}_{[\![s_2]\!]_{\mu_{|\neg e}}})\}.$$

  We apply the [Conseq] rule to be able to apply the [Cond] rule with $\eta_1 = \mathbf{1}_{[\![s_1]\!]_{\mu_{|e}}}$ and $\eta_2 = \mathbf{1}_{[\![s_2]\!]_{\mu_{|\neg e}}}$ Both premises can be proved by an application of the [Conseq] rule followed by the application of the induction hypothesis.
- $s = \mathbf{while}\ e\ \mathbf{do}\ s$, we have to prove

  $$\{\mathbf{1}_\mu\}\, \mathbf{while}\ e\ \mathbf{do}\ s\, \{\mathbf{1}_{\lim_{n\to\infty}\ [\![(\mathbf{if}\ e\ \mathbf{then}\ s)^n; \mathbf{if}\ e\ \mathbf{then}\ \mathbf{abort}]\!]_\mu}\}.$$

  We first apply the [While] rule with $\eta'_n = \mathbf{1}_{[\![(\mathbf{if}\ e\ \mathbf{then}\ s)^n]\!]_\mu}$ and

  $$\eta_n = \mathbf{1}_{[\![(\mathbf{if}\ e\ \mathbf{then}\ s)^n; \mathbf{if}\ e\ \mathbf{then}\ \mathbf{abort}]\!]_\mu}.$$

  For the first premise we apply the same process as for the conditional case: we apply the [Conseq] and [Cond] rules and we conclude using the induction hypothesis (and the [Skip] rule). For the second premise we follow the same process but we conclude using the [Abort] rule instead of the induction hypothesis. Finally we conclude since $\mathsf{uclosed}((\eta_n)_{n\in\mathbb{N}^\infty})$. □

The abstract logic is also relatively complete. This property will be less important for our purposes, but it serves as a basic sanity check.

Theorem 2 (Relative completeness). *Every valid judgment is derivable.*

Proof. Consider a valid judgment $\{\eta\}s\{\eta'\}$. Let μ be a probabilistic state such that $\eta(\mu)$. By the above proposition, $\{\mathbf{1}_\mu\}s\{\mathbf{1}_{[\![s]\!]_\mu}\}$. Using the validity of the judgment and [Conseq], we have $\{\mathbf{1}_\mu \wedge \eta(\mu)\}s\{\eta'\}$. Using the [Exists] and [Conseq] rules, we conclude $\{\eta\}s\{\eta'\}$ as required. □

The side-conditions in the loop rules (e.g., uclosed/tclosed/dclosed and the weight conditions) are difficult to prove, since they are semantic properties. Next, we present a concrete version of the logic with give easy-to-check, syntactic sufficient conditions.

5 A Concrete Program Logic

To give a more practical version of the logic, we begin by setting a concrete syntax for assertions

Assertions. We use a two-level assertion language, presented in Fig. 5. A *probabilistic assertion* η is a formula built from comparison of probabilistic expressions, using first-order quantifiers and connectives, and the special connective $\oplus$. A *probabilistic expression* p can be a logical variable v, an operator applied to probabilistic expressions $o(\boldsymbol{p})$ (constants are 0-ary operators), or the expectation $\mathbb{E}[\tilde{e}]$ of a state expression $\tilde{e}$. A *state expression* $\tilde{e}$ is either a program variable x, the characteristic function $\mathbf{1}_\phi$ of a state assertion ϕ, an operator applied to state expressions $o(\tilde{\boldsymbol{e}})$, or the expectation $\mathbb{E}_{v\sim g}[\tilde{e}]$ of state expression $\tilde{e}$ in a given distribution g. Finally, a *state assertion* ϕ is a first-order formula over program variables. Note that the set of operators is left unspecified but we assume that all the expressions in $\mathcal{E}$ and $\mathcal{D}$ can be encoded by operators.

$$
\begin{array}{llr}
\tilde{e} ::= x \mid v \mid \mathbf{1}_\phi \mid \mathbb{E}_{v\sim g}[\tilde{e}] \mid o(\tilde{\boldsymbol{e}}) & & \text{(S-expr.)}\\
\phi ::= \tilde{e} \bowtie \tilde{e} \mid FO(\phi) & & \text{(S-assn.)}\\
p ::= v \mid o(\boldsymbol{p}) \mid \mathbb{E}[\tilde{e}] & & \text{(P-expr.)}\\
\eta ::= p \bowtie p \mid \eta \oplus \eta \mid FO(\eta) & & \text{(P-assn.)}\\
\bowtie\, \in \{=, <, \leq\} & o \in Ops & \text{(Ops.)}
\end{array}
$$

Fig. 5. Assertion syntax

The interpretation of the concrete syntax is as expected. The interpretation of probabilistic assertions is relative to a valuation ρ which maps logical variables to values, and is an element of Assn. The definition of the interpretation is straightforward; the only interesting case is $[\![\mathbb{E}[\tilde{e}]]\!]^\rho_\mu$ which is defined by $\mathbb{E}_{m\sim\mu}[[\![\tilde{e}]\!]^\rho_m]$, where $[\![\tilde{e}]\!]^\rho_m$ is the interpretation of the state expression $\tilde{e}$ in the memory m and valuation ρ. The interpretation of state expressions is a mapping from memories to values, which can be lifted to a mapping from distributions over memories to distributions over values. The definition of the interpretation is straightforward; the most interesting case is for expectation $[\![\mathbb{E}_{v\sim g}[\tilde{e}]]\!]^\rho_m \triangleq \mathbb{E}_{w\sim[\![g]\!]^\rho_m}[[\![\tilde{e}]\!]^{\rho[v:=w]}_m]$. We present the full interpretations in the supplemental materials.

Many standard concepts from probability theory have a natural representation in our syntax. For example:

- the probability that ϕ holds in some probabilistic state is represented by the probabilistic expression $\Pr[\phi] \triangleq \mathbb{E}[\mathbf{1}_\phi]$;
- probabilistic independence of state expressions $\tilde{e}_1, \ldots, \tilde{e}_n$ is modeled by the probabilistic assertion $\#\{\tilde{e}_1, \ldots, \tilde{e}_n\}$, defined by the clause[4]

$$\forall v_1 \ldots v_n,\ \Pr[\top]^{n-1}\Pr[\bigwedge_{i=1\ldots n} \tilde{e}_i = v_i] = \prod_{i=1\ldots n} \Pr[\tilde{e}_i = v_i];$$

- the fact that a distribution is proper is modeled by the probabilistic assertion $\mathcal{L} \triangleq \Pr[\top] = 1$;

[4] The term $\Pr[\top]^{n-1}$ is necessary since we work with sub-distributions.

– a state expression $\tilde{e}$ distributed according to a law g is modeled by the probabilistic assertion

$$\tilde{e} \sim g \triangleq \forall w,\ \ \Pr[\tilde{e} = w] = \mathbb{E}[\mathbb{E}_{v\sim g}[\mathbf{1}_{v=w}]].$$

The inner expectation computes the probability that v drawn from g is equal to a fixed w; the outer expectation weights the inner probability by the probability of each value of w.

We can easily define $\Box$ operator from the previous section in our new syntax: $\Box\phi \triangleq \Pr[\neg\phi] = 0$.

Syntactic Proof Rules. Now that we have a concrete syntax for assertions, we can give syntactic versions of many of the existing proof rules. Such proof rules are often easier to use since they avoid reasoning about the semantics of commands and assertions. We tackle the non-looping rules first, beginning with the following syntactic rules for assignment and sampling:

$$\frac{}{\{\eta[x := e]\}x \leftarrow e\{\eta\}}\ [\textsc{Assgn}] \qquad\qquad \frac{}{\{\mathcal{P}^g_x(\eta)\}x \xleftarrow{\$} g\{\eta\}}\ [\textsc{Sample}]$$

The rule for assignment is the usual rule from Hoare logic, replacing the program variable x by its corresponding expression e in the pre-condition. The replacement $\eta[x := e]$ is done recursively on the probabilistic assertion η; for instance for expectations, it is defined by $\mathbb{E}[\tilde{e}][x := e] \triangleq \mathbb{E}[\tilde{e}[x := e]]$, where $\tilde{e}[x := e]$ is the syntactic substitution.

The rule for sampling uses probabilistic substitution operator $\mathcal{P}^g_x(\eta)$, which replaces all occurrences of x in η by a new integration variable t and records that t is drawn from g; the operator is defined in Fig. 6.

$$\begin{aligned}
\mathcal{P}^g_x(v) &\triangleq v\\
\mathcal{P}^g_x(\mathbb{E}[\tilde{e}]) &\triangleq \mathbb{E}[\mathbb{E}_{t\sim g}[\tilde{e}[x := t]]]\\
\mathcal{P}^g_x(o(\boldsymbol{\eta})) &\triangleq o(\mathcal{P}^g_x(\eta_1), \dots, \mathcal{P}^g_x(\eta_n))\\
\mathcal{P}^g_x(\eta_1 \bowtie \eta_2) &\triangleq \mathcal{P}^g_x(\eta_1) \bowtie \mathcal{P}^g_x(\eta_2)
\end{aligned}$$

for $o \in \mathbf{Ops}, \bowtie \in \{\wedge, \vee, \Rightarrow\}$.

Fig. 6. Syntactic op. $\mathcal{P}$ (main cases)

Next, we turn to the loop rule. The side-conditions from Fig. 3 are purely semantic, while in practice it is more convenient to use a sufficient condition in the Hoare logic. We give sufficient conditions for ensuring certain and almost-sure termination in Fig. 7; $\tilde{e}$ is an integer-valued expression. The first side-condition $\mathcal{C}_{\text{CTerm}}$ shows certain termination given a strictly decreasing *variant* $\tilde{e}$ that is bounded below, similar to how a decreasing variant shows termination for deterministic programs. The second side-condition $\mathcal{C}_{\text{ASTerm}}$ shows almost-sure termination given a probabilistic variant $\tilde{e}$, which must be bounded both above and below. While $\tilde{e}$ may increase with some probability, it must decrease with strictly positive probability. This condition was previously considered by [17] for probabilistic transition systems and also used in expectation-based approaches [20,33]. Our framework can also support more refined conditions (e.g., based on super-martingales [9,31]), but the condition $\mathcal{C}_{\text{ASTerm}}$ already suffices for most randomized algorithms.

$$\begin{aligned}\mathcal{C}_{\text{CTerm}} &\triangleq \{\mathcal{L} \wedge \Box(\tilde{e} = k \wedge 0 < k \wedge b)\}\ s\ \{\mathcal{L} \wedge \Box(\tilde{e} < k)\}\\ &\models \eta \Rightarrow (\exists \dot{y}.\ \Box\tilde{e} \leq \dot{y}) \wedge \Box(\tilde{e} = 0 \Rightarrow \neg b)\\ \mathcal{C}_{\text{ASTerm}} &\triangleq \{\mathcal{L} \wedge \Box(\tilde{e} = k \wedge 0 < k \leq K \wedge b)\}\ s\ \{\mathcal{L} \wedge \Box(0 \leq \tilde{e} \leq K) \wedge \Pr[\tilde{e} < k] \geq \epsilon\}\\ &\models \eta \Rightarrow \Box(0 \leq \tilde{e} \leq K \wedge \tilde{e} = 0 \Rightarrow \neg b)\\ &\models \mathsf{tclosed}(\eta)\end{aligned}$$

Fig. 7. Side-conditions for loop rules

While t-closedness is a semantic condition (cf. Definition 10), there are simple syntactic conditions to guarantee it. For instance, assertions that carry a non-strict comparison $\bowtie \in \{\leq, \geq, =\}$ between two bounded probabilistic expressions are t-closed; the assertion stating probabilistic independence of a set of expressions is t-closed.

Precondition Calculus. With a concrete syntax for assertions, we are also able to incorporate syntactic reasoning principles. One classic tool is Morgan and McIver's *greatest pre-expectation*, which we take as inspiration for a pre-condition calculus for the loop-free fragment of ELLORA. Given an assertion η and a loop-free statement s, we mechanically construct an assertion η^* that is the pre-condition of s that implies η as a post-condition. The basic idea is to replace each expectation expression p inside η by an expression p^* that has the same denotation before running s as p after running s. This process yields an assertion η^* that, interpreted before running s, is logically equivalent to η interpreted after running s.

The computation rules for pre-conditions are defined in Fig. 8. For a probability assertion η, its pre-condition $\mathrm{pc}(s, \eta)$ corresponds to η where the expectation expressions of the form $\mathbb{E}[\tilde{e}]$ are replaced by their corresponding *pre-term*, $\mathrm{pe}(s, \mathbb{E}[\tilde{e}])$. Pre-terms correspond loosely to Morgan and McIver's *pre-expectations*—we will make this correspondence more precise in the next section. The main interesting cases for computing pre-terms are for random sampling and conditionals. For random sampling the result is $\mathcal{P}_x^g(\mathbb{E}[\tilde{e}])$, which corresponds to the [SAMPLE] rule. For conditionals, the expectation expression is split into a part where e is true and a part where e is not true. We restrict the expectation to a part satisfying e with the operator $\mathbb{E}[\tilde{e}]_{|e} \triangleq \mathbb{E}[\tilde{e} \cdot \mathbf{1}_e]$. This corresponds to the expected value of $\tilde{e}$ on the portion of the distribution where e is true. Then, we can build the pre-condition calculus into ELLORA.

Theorem 1. *Let s be a non-looping command. Then, the following rule is derivable in the concrete version of* ELLORA*:*

$$\frac{}{\{pc(s, \eta)\}\ s\ \{\eta\}}\ [\text{PC}]$$

6 Case Studies: Embedding Lightweight Logics

While ELLORA is suitable for general-purpose reasoning about probabilistic programs, in practice humans typically use more special-purpose proof

$$\begin{aligned}
\mathrm{pe}(s_1; s_2, \mathbb{E}[\tilde{e}]) &\triangleq \mathrm{pe}(s_1, \mathrm{pe}(s_2, \mathbb{E}[\tilde{e}])) \\
\mathrm{pe}(x \leftarrow e, \mathbb{E}[\tilde{e}]) &\triangleq \mathbb{E}[\tilde{e}][x := e] \\
\mathrm{pe}(x \xleftarrow{\$} g, \mathbb{E}[\tilde{e}]) &\triangleq \mathcal{P}_x^g(\mathbb{E}[\tilde{e}]) \\
\mathrm{pe}(\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \mathbb{E}[\tilde{e}]) &\triangleq \mathrm{pe}(s_1, \mathbb{E}[\tilde{e}])_{|e} + \mathrm{pe}(s_2, \mathbb{E}[\tilde{e}])_{|\neg e} \\
\\
\mathrm{pc}(s, p_1 \bowtie p_2) &\triangleq \mathrm{pe}(s, p_1) \bowtie \mathrm{pe}(s, p_2)
\end{aligned}$$

Fig. 8. Precondition calculus (selected)

techniques—often targeting just a single, specific kind of property, like probabilistic independence—when proving probabilistic assertions. When these techniques apply, they can be a convenient and powerful tool.

To capture this intuitive style of reasoning, researchers have considered lightweight program logics where the assertions and proof rules are tailored to a specific proof technique. We demonstrate how to integrate these tools in an assertion-based logic by introducing and embedding a new logic for reasoning about independence and distribution laws, useful properties when analyzing randomized algorithms. We crucially rely on the rich assertions in Ellora—it is not clear how to extend expectation-based approaches to support similar, lightweight reasoning. Then, we show to embed the union bound logic [4] for proving accuracy bounds.

6.1 Law and Independence Logic

We begin by describing the law and independence logic IL, a proof system with intuitive rules that are easy to apply and amenable to automation. For simplicity, we only consider programs which sample from the binomial distribution, and have deterministic control flow—for lack of space, we also omit procedure calls.

Definition 12 (Assertions). IL *assertions have the grammar:*

$$\xi := \mathsf{det}(e) \mid \#E \mid e \sim \mathrm{B}(e, p) \mid \top \mid \bot \mid \xi \wedge \xi$$

where $e \in \mathcal{E}$, $E \subseteq \mathcal{E}$, *and* $p \in [0, 1]$.

The assertion $\mathsf{det}(e)$ states that e is deterministic in the current distribution, i.e., there is at most one element in the support of its interpretation. The assertion $\#E$ states that the expressions in E are independent, as formalized in the previous section. The assertion $e \sim \mathrm{B}(m, p)$ states that e is distributed according to a binomial distribution with parameter m (where m can be an expression) and constant probability p, i.e. the probability that $e = k$ is equal to the probability that exactly k independent coin flips return heads using a biased coin that returns heads with probability p.

Assertions can be seen as an instance of a logical abstract domain, where the order between assertions is given by implication based on a small number of axioms. Examples of such axioms include independence of singletons, irreflexivity of independence, anti-monotonicity of independence, an axiom for the sum of binomial distributions, and rules for deterministic expressions:

$$\#\{x\} \qquad \#\{x,x\} \iff \mathsf{det}(x) \qquad \#(E \cup E') \implies \#E$$

$$e \sim \mathrm{B}(m,p) \wedge e' \sim \mathrm{B}(m',p) \wedge \# \ \{e,e'\} \implies e+e' \sim \mathrm{B}(m+m',p)$$

$$\bigwedge_{1\leq i\leq n} \mathsf{det}(e_i) \implies \mathsf{det}(f(e_1,\ldots,e_n))$$

Definition 13. *Judgments of the logic are of the form* $\{\xi\} \ \ s \ \ \{\xi'\}$*, where* ξ *and* ξ' *are* IL*-assertions. A judgment is* valid *if it is derivable from the rules of Fig. 9; structural rules and rule for sequential composition are similar to those from Sect. 4 and omitted.*

The rule [IL-Assgn] for deterministic assignments is as in Sect. 4. The rule [IL-Sample] for random assignments yields as post-condition that the variable x and a set of expressions E are independent assuming that E is independent before the sampling, and moreover that x follows the law of the distribution that it is sampled from. The rule [IL-Cond] for conditionals requires that the guard is deterministic, and that each of the branches satisfies the specification; if the guard is not deterministic, there are simple examples where the rule is not sound. The rule [IL-While] for loops requires that the loop is certainly terminating with a deterministic guard. Note that the requirement of certain termination could be avoided by restricting the structural rules such that a statement s has deterministic control flow whenever $\{\xi\} \ \ s \ \ \{\xi'\}$ is derivable.

We now turn to the embedding. The embedding of IL assertions into general assertions is immediate, except for $\mathsf{det}(e)$ which is translated as $\Box e \vee \Box \neg e$. We let $\overline{\xi}$ denote the translation of ξ.

Theorem 2 (Embedding and soundness of IL logic)**.** *If* $\{\xi\} \ \ s \ \ \{\xi'\}$ *is derivable in the* IL *logic, then* $\{\overline{\xi}\}\, s\, \{\overline{\xi'}\}$ *is derivable in (the syntactic variant of)* Ellora*. As a consequence, every derivable judgment* $\{\xi\} \ \ s \ \ \{\xi'\}$ *is valid.*

Proof sketch. By induction on the derivation. The interesting cases are conditionals and loops. For conditionals, the soundness follows from the soundness of the rule:

$$\frac{\{\eta\}\, s_1\, \{\eta'\} \qquad \{\eta\}\, s_2\, \{\eta'\} \qquad \Box e \vee \Box\neg e}{\{\eta\}\, \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2\, \{\eta'\}}$$

To prove the soundness of this rule, we proceed by case analysis on $\Box e \vee \Box\neg e$. We treat the case $\Box e$; the other case is similar. In this case, η is equivalent to $\eta_1 \wedge \Box e \oplus \eta_2 \wedge \Box\neg e$, where $\eta_1 = \eta$ and $\eta_2 = \bot$. Let $\eta_1' = \eta'$ and $\eta_2 = \Box\bot$; again, $\eta_1' \oplus \eta_2'$ is logically equivalent to η'. The soundness of the rule thus follows from

$$\frac{}{\{\xi[x := e]\}\ x \leftarrow e\ \{\xi\}}\ [\text{IL-Assgn}]$$

$$\frac{\{x\} \cap \mathrm{FV}(E) \cap \mathrm{FV}(e) = \emptyset}{\{\# E\}\ x \overset{\$}{\leftarrow} \mathrm{B}(e,p)\ \{\#(E \cup \{x\}) \wedge x \sim \mathrm{B}(e,p)\}}\ [\text{IL-Sample}]$$

$$\frac{\{\xi\}\ s_1\ \{\xi'\} \qquad \{\xi'\}\ s_2\ \{\xi''\}}{\{\xi\}\ s_1; s_2\ \{\xi''\}}\ [\text{IL-Seq}]$$

$$\frac{\begin{array}{c}\{\xi\}\ s_1\ \{\xi'\} \qquad \{\xi\}\ s_2\ \{\xi'\} \\ \xi \implies \mathsf{det}(b)\end{array}}{\{\xi\}\ \mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \{\xi'\}}\ [\text{IL-Cond}]$$

$$\frac{\{\xi\}\ s\ \{\xi\} \qquad \xi \implies \mathsf{det}(b) \qquad \mathcal{C}_{\text{CTerm}}}{\{\xi\}\ \mathbf{while}\ b\ \mathbf{do}\ s\ \{\xi\}}\ [\text{IL-While}]$$

Fig. 9. IL proof rules (selected)

the soundness of the [Cond] and [Conseq] rules. For loops, there exists a natural number n such that **while** b **do** s is semantically equivalent to $(\mathbf{if}\ b\ \mathbf{then}\ s)^n$. By assumption $\{\xi\}\ s\ \{\xi\}$ holds, and thus by induction hypothesis $\{\overline{\xi}\}\, s\, \{\overline{\xi}\}$. We also have $\xi \implies \mathsf{det}(b)$, and hence $\{\overline{\xi}\}\ \mathbf{if}\ b\ \mathbf{then}\ s\, \{\overline{\xi}\}$. We conclude by [Seq]. □

To illustrate our system IL, consider the statement s in Fig. 10 which flips a fair coin N times and counts the number of heads. Using the logic, we prove that $\mathtt{c} \sim \mathrm{B}(N \cdot (N+1)/2, 1/2)$ is a post-condition for s. We take the invariant:

$$\mathtt{c} \sim \mathrm{B}\,(\mathtt{j}(\mathtt{j}+1)/2, 1/2)$$

The invariant holds initially, as $0 \sim B(0, 1/2)$. For the inductive case, we show:

$$\{\mathtt{c} \sim \mathrm{B}\,(0, 1/2)\}\ s_0\ \{\mathtt{c} \sim \mathrm{B}\,((\mathtt{j}+1)(\mathtt{j}+2)/2, 1/2)\}$$

where s_0 represents the loop body, i.e. $\mathtt{x} \overset{\$}{\leftarrow} \mathrm{B}\,(\mathtt{j}, 1/2)\,;\mathtt{c} \leftarrow \mathtt{c} + \mathtt{x}$. First, we apply the rule for sequence taking as intermediate assertion

$$\mathtt{c} \sim \mathrm{B}\,(\mathtt{j}(\mathtt{j}+1)/2, 1/2) \wedge \mathtt{x} \sim \mathrm{B}\,(\mathtt{j}, 1/2) \wedge \#\{\mathtt{x}, \mathtt{c}\}$$

```
proc sum () =
  var c:int, x:int;
  c ← 0;
  for j ← 1 to N do
    x ←$ B(j,1/2);
    c ← c + x;
  return c
```

Fig. 10. Sum of bin.

The first premise follows from the rule for random assignment and structural rules. The second premise follows from the rule for deterministic assignment and the rule of consequence, applying axioms about sums of binomial distributions.

We briefly comment on several limitations of IL. First, IL is restricted to programs with deterministic control flow, but this restriction could be partially relaxed by enriching IL with assertions for conditional independence. Such assertions are already expressible in the logic of Ellora; adding conditional independence would significantly broaden the scope of the IL proof system and open the possibility to rely on axiomatizations of conditional independence (e.g., based on graphoids [36]). Second, the logic only supports sampling from binomial distributions. It is possible to enrich the language of assertions with clauses $c \sim g$ where g can model other distributions, like the uniform distribution or the Laplace distribution. The main design challenge is finding a core set of useful facts about these distributions. Enriching the logic and automating the analysis are interesting avenues for further work.

6.2 Embedding the Union Bound Logic

The program logic aHL [4] was recently introduced for estimating accuracy of randomized computations. One main application of aHL is proving accuracy of randomized algorithms, both in the offline and online settings—i.e. with adversary calls. aHL is based on the union bound, a basic tool from probability theory, and has judgments of the form $\models_\beta \{\Phi\}\ s\ \{\Psi\}$, where s is a statement, Φ and Ψ are first-order formulae over program variables, and β is a probability, i.e. $\beta \in [0,1]$. A judgment $\models_\beta \{\Phi\}\ s\ \{\Psi\}$ is valid if for every memory m such that $\Phi(m)$, the probability of $\neg\Psi$ in $[\![s]\!]_m$ is upper bounded by β, i.e. $\Pr_{[\![s]\!]_m}[\neg\Psi] \leq \beta$.

Figure 11 presents some key rules of aHL, including a rule for sampling from the Laplace distribution $\mathcal{L}_\epsilon$ centered around e. The predicate $\mathcal{C}_{\text{CTerm}}(k)$ indicates that the loop terminates in at most k steps on any memory that satisfies the pre-condition. Moreover, β is a function of ϵ.

$$\frac{}{\models_\beta \{\top\}\ x \xleftarrow{\$} \mathcal{L}_\epsilon(e)\ \{|x - e| \leq \frac{1}{\epsilon}\log\frac{1}{\beta}\}}\ [\text{aHL-Sample}]$$

$$\frac{\models_{\beta_1} \{\Phi\}\ s_1\ \{\Theta\} \qquad \models_{\beta_2} \{\Theta\}\ s_2\ \{\Psi\}}{\models_{\beta_1+\beta_2} \{\Phi\}\ s_1; s_2\ \{\Psi\}}\ [\text{aHL-Seq}]$$

$$\frac{\models_\beta \{\Phi\}\ c\ \{\Phi\} \qquad \mathcal{C}_{\text{CTerm}}(k)}{\models_{k\cdot\beta} \{\Phi\}\ \mathbf{while}\ e\ \mathbf{do}\ c\ \{\Phi \wedge \neg e\}}\ [\text{aHL-While}]$$

Fig. 11. aHL proof rules (selected)

aHL has a simple embedding into Ellora.

Theorem 3 (Embedding of aHL). *If $\models_\beta \{\Phi\}\ s\ \{\Psi\}$ is derivable in* aHL*, then $\{\Box\Phi\}\ s\ \{\mathbb{E}[\mathbf{1}_{\neg\Psi}] \leq \beta\}$ is derivable in* Ellora.

7 Case Studies: Verifying Randomized Algorithms

In this section, we will demonstrate Ellora on a selection of examples; we present further examples in the supplemental material. Together, they exhibit a wide variety of different proof techniques and reasoning principles which are available in the ELLORA's implementation.

Hypercube Routing. will begin with the *hypercube routing* algorithm [41,42]. Consider a network topology (the *hypercube*) where each node is labeled by a bitstring of length D and two nodes are connected by an edge if and only if the two corresponding labels differ in exactly one bit position.

In the network, there is initially one packet at each node, and each packet has a unique destination. The algorithm implements a routing strategy based on *bit fixing*: if the current position has bitstring i, and the target node has bitstring j, we compare the bits in i and j from left to right, moving along the edge that corrects the first differing bit. Valiant's algorithm uses randomization to guarantee that the total number of steps grows *logarithmically* in the number of packets. In the first phase, each packet i select an intermediate destination $\rho(i)$ uniformly at random, and use bit fixing to reach $\rho(i)$. In the second phase, each packet use bit fixing to go from $\rho(i)$ to the destination j. We will focus on the first phase since the reasoning for the second phase is nearly identical. We can model the strategy with the code in Fig. 12, using some syntactic sugar for the **for** loops.[5]

```
proc route ($D$ $T$ : int) :
  var $\rho$, pos, usedBy : node map;
  var nextE : edge;
  pos ← Map.init id $2^D$; $\rho$ ←Map.empty;
  for $i \leftarrow$ 1 to $2^D$ do
    $\rho$[i] $\overset{\$}{\leftarrow}$[1, $2^D$]
    for $t \leftarrow$ 1 to $T$ do
      usedBy ← Map.empty;
      for $i \leftarrow$ 1 to $2^D$ do
        if pos[$i$] $\neq$ $\rho$[$i$] then
          nextE ← getEdge pos[$i$] $\rho$[$i$];
          if usedBy[nextE] = $\perp$ then
            // Mark edge used
            usedBy[nextE] ← $i$;
            // Move packet
            pos[$i$] ← dest nextE
  return (pos, $\rho$)
```

Fig. 12. Hypercube Routing

We assume that initially, the position of the packet i is at node i (see `Map.init`). Then, we initialize the random intermediate destinations ρ. The remaining loop encodes the evaluation of the routing strategy iterated T time. The variable `usedBy` is a map that logs if an edge is already used by a packet, it is empty at the beginning of each iteration. For each packet, we try to move it across one edge along the path to its intermediate destination. The function `getEdge` returns the next edge to follow, following the bit-fixing scheme. If the packet can progress (its edge is not used), then its current position is updated and the edge is marked as used.

We show that if the number of timesteps T is $4D + 1$, then all packets reach their intermediate destination in at most T steps, except with a small probability 2^{-2D} of failure. That is, the number of timesteps grows linearly in D, logarithmic in the number of packets. This is formalized in our system as:

$$\{T = 4D + 1\}\texttt{route}\{\Pr[\exists i.\ \texttt{pos}[i] \neq \rho[i]] \leq 2^{-2D}]\}$$

[5] Recall that the number of node in a hypercube of dimension D is 2^D so each node can be identified by a number in $[1, 2^D]$.

```
proc coupon ($N$ : int) :
  var int cp[$N$], $t$[$N$];
  var int $X \leftarrow$0;
  for $p \leftarrow$ 1 to $N$ do
    ct $\leftarrow$ 0;
    cur $\xleftarrow{\$}$ [1, $N$];
    while cp[cur] = 1 do
      ct $\leftarrow$ ct + 1;
      cur $\xleftarrow{\$}$ [1, $N$];
    $t$[$p$]    $\leftarrow$ ct;
    cp[cur] $\leftarrow$ 1;
    $X \leftarrow X$ + $t$[$p$];
  return $X$
```

Fig. 13. Coupon collector

Modeling Infinite Processes. Our second example is the *coupon collector* process. The algorithm draws a uniformly random coupon (we have N coupon) on each day, terminating when it has drawn at least one of each kind of coupon. The code of the algorithm is displayed in Fig. 13; the array `cp` records of the coupons seen so far, `t` holds the number of steps taken before seeing a new coupon, and X tracks of the total number of steps. Our goal is to bound the average number of iterations. This is formalized in our logic as:

$$\{\mathcal{L}\}\ \texttt{coupon}\ \left\{\mathbb{E}[X] = \sum_{i\in[1,N]}\left(\frac{N}{N-i+1}\right)\right\}.$$

Limited Randomness. Pairwise independence says that if we see the result of X_i, we do not gain information about all other variables X_k. However, if we see the result of *two* variables X_i, X_j, we may gain information about X_k. There are many constructions in the algorithms literature that grow a small number of independent bits into more pairwise independent bits. Figure 14 gives one procedure, where $\oplus$ is exclusive-or, and `bits(j)` is the set of positions set to 1 in the binary expansion of `j`. The proof uses the following fact, which we fully verify: for a uniformly distributed Boolean random variable Y, and a random variable Z of any type,

```
proc pwInd (N : int) :
  var bool X[$2^N$], B[N];
  for i $\leftarrow$ 1 to N do
    B[i] $\xleftarrow{\$}$ Ber(1/2);
  for j $\leftarrow$ 1 to $2^N$ do
    X[j] $\leftarrow$ 0;
    for k $\leftarrow$ 1 to N do
      if k $\in$ bits(j) then
        X[j] $\leftarrow$ X[j] $\oplus$ B[k]
  return X
```

Fig. 14. Pairwise Independence

$$Y \# Z \Rightarrow Y \oplus f(Z) \# g(Z) \tag{1}$$

for any two Boolean functions f, g. Then, note that $\texttt{X}[i] = \bigoplus_{\{j\in\texttt{bits}(i)\}} \texttt{B}[j]$ where the big XOR operator ranges over the indices j where the bit representation of i has bit j set. For any two $i, k \in [1, \ldots, 2^{\texttt{N}}]$ distinct, there is a bit position in $[1, \ldots, \texttt{N}]$ where i and k differ; call this position r and suppose it is set in i but not in k. By rewriting,

$$\texttt{X}[i] = \texttt{B}[r] \oplus \bigoplus_{\{j\in\texttt{bits}(i)\setminus r\}} \texttt{B}[j] \quad \text{and} \quad \texttt{X}[k] = \bigoplus_{\{j\in\texttt{bits}(k)\setminus r\}} \texttt{B}[j].$$

Since $\texttt{B}[j]$ are all independent, $\texttt{X}[i] \# \texttt{X}[k]$ follows from Eq. (1) taking Z to be the distribution on tuples $\langle \texttt{B}[1], \ldots, \texttt{B}[\texttt{N}]\rangle$ excluding $\texttt{B}[r]$. This verifies pairwise independence:

$$\{\mathcal{L}\}\ \texttt{pwInd(N)}\ \{\mathcal{L} \wedge \forall i, k \in [2^{\texttt{N}}].\ i \neq k \Rightarrow \texttt{X}[i] \# \texttt{X}[k]\}.$$

Adversarial Programs. Pseudorandom functions (PRF) and pseudorandom permutations (PRP) are two idealized primitives that play a central role in the design of symmetric-key systems. Although the most natural assumption to make about a blockcipher is that it behaves as a pseudorandom permutation, most commonly the security of such a system is analyzed by replacing the blockcipher with a perfectly random function. The PRP/PRF Switching Lemma [6,22] fills the gap: given a bound for the security of a blockcipher as a pseudorandom function, it gives a bound for its security as a pseudorandom permutation.

Lemma 4 (PRP/PRF switching lemma). *Let A be an adversary with blackbox access to an oracle O implementing either a random permutation on $\{0,1\}^l$ or a random function from $\{0,1\}^l$ to $\{0,1\}^l$. Then the probability that the adversary A distinguishes between the two oracles in at most q calls is bounded by*

$$|\Pr_{PRP}[b \wedge |H| \leq q] - \Pr_{PRF}[b \wedge |H| \leq q]| \leq \frac{q(q-1)}{2^{l+1}},$$

where H is a map storing each adversary call and $|H|$ is its size.

Proving this lemma can be done using the Fundamental Lemma of Game-Playing, and bounding the probability of *bad* in the program from Fig. 15. We focus on the latter. Here we apply the [Adv] rule of Ellora with the invariant $\forall k, \Pr[\mathtt{bad} \wedge |H| \leq k] \leq \frac{k(k-1)}{2^{l+1}}$ where $|H|$ is the size of the map H, i.e. the number of adversary call. Intuitively, the invariant says that at each call to the oracle the probability that bad has been set before and that the number of adversary call is less than k is bounded by a polynomial in k.

The invariant is d-closed and true before the adversary call, since at that point $\Pr[\mathtt{bad}] = 0$. Then we need to prove that the oracle preserves the invariant, which can be done easily using the precondition calculus ([PC] rule).

```
var H: ({0,1}^l, {0,1}^l) map;
proc orcl (q:{0,1}^l):
  var a : {0,1}^l;
  if q ∉ H then
    a  ←$ {0,1}^l;
    bad ← bad || a ∈ codom(H);
    H[q] ← a;
  return H[q];

proc main():
  var b: bool;
  bad ← false;
  H   ← [];
  b   ← A();
  return b;
```

Fig. 15. PRP/PRF game

8 Implementation and Mechanization

We have built a prototype implementation of Ellora within EasyCrypt [2,5], a theorem prover originally designed for verifying cryptographic protocols. EasyCrypt provides a convenient environment for constructing proofs in various Hoare logics, supporting interactive, tactic-based proofs for manipulating assertions and allowing users to invoke external tools, like SMT-solvers, to discharge proof obligations. EasyCrypt provides a mature set of libraries for both data structures (sets, maps, lists, arrays, etc.) and mathematical theorems (algebra, real analysis, etc.), which we extended with theorems from probability theory.

Table 1. Benchmarks

Example	LC	FPLC
`hypercube`	100	1140
`coupon`	27	184
`vertex-cover`	30	61
`pairwise-indep`	30	231
`private-sums`	22	80
`poly-id-test`	22	32
`random-walk`	16	42
`dice-sampling`	10	64
`matrix-prod-test`	20	75

We used the implementation for verifying many examples from the literature, including all the programs presented in Sect. 7 as well as some additional examples in Table 1 (such as polynomial identity test, private running sums, properties about random walks, etc.). The verified proofs bear a strong resemblance to the existing, paper proofs. Independently of this work, Ellora has been used to formalize the main theorem about a randomized gossip-based protocol for distributed systems [26, Theorem 2.1]. Some libraries developed in the scope of Ellora have been incorporated into the main branch of EasyCrypt, including a general library on probabilistic independence.

A New Library for Probabilistic Independence. In order to support assertions of the concrete program logic, we enhanced the standard libraries of EasyCrypt, notably the ones dealing with big operators and sub-distributions. Like all EasyCrypt libraries, they are written in a foundational style, i.e. they are defined instead of axiomatized. A large part of our libraries are proved formally from first principles. However, some results, such as concentration bounds, are currently declared as axioms.

Our formalization of probabilistic independence deserves special mention. We formalized two different (but logically equivalent) notions of independence. The first is in terms of products of probabilities, and is based on heterogenous lists. Since Ellora (like EasyCrypt) has no support for heterogeneous lists, we use a smart encoding based on second-order predicates. The second definition is more abstract, in terms of product and marginal distributions. While the first definition is easier to use when reasoning about randomized algorithms, the second definition is more suited for proving mathematical facts. We prove the two definitions equivalent, and formalize a collection of related theorems.

Mechanized Meta-Theory. The proofs of soundness and relative completeness of the abstract logic, without adversary calls, and the syntactical termination arguments have been mechanized in the Coq proof assistant. The development is available in supplemental material.

9 Related Work

More on Assertion-Based Techniques. The earliest assertion-based system is due to Ramshaw [37], who proposes a program logic where assertions can be formulas involving *frequencies*, essentially probabilities on sub-distributions. Ramshaw's logic allows assertions to be combined with operators like $\oplus$, similar to our approach. [18] presents a Hoare-style logic with general assertions on the distribution, allowing expected values and probabilities. However, his **while** rule is based on a semantic condition on the guarded loop body, which is less desirable for verification because it requires reasoning about the semantics of programs. [8] give decidability results for a probabilistic Hoare logic without **while** loops. We are not aware of any existing system that supports assertions about general expected values; existing works also restrict to Boolean distributions. [38] formalize a Hoare logic for probabilistic programs but unlike our work, their assertions are interpreted on *distributions* rather than sub-distributions. For conditionals, their semantics rescales the distribution of states that enter each branch. However, their assertion language is limited and they impose strong restrictions on loops.

Other Approaches. Researchers have proposed many other approaches to verify probabilistic program. For instance, verification of Markov transition systems goes back to at least [17,40]; our condition for ensuring almost-sure termination in loops is directly inspired by their work. Automated methods include model checking (see e.g., [1,25,29]) and abstract interpretation (see e.g., [12,32]). Techniques for reasoning about higher-order (functional) probabilistic languages are an active subject of research (see e.g., [7,13,14]). For analyzing probabilistic loops, in particular, there are tools for reasoning about running time. There are also automated systems for synthesizing invariants [3,11]. [9,10] use a martingale method to compute the expected time of the coupon collector process for $N = 5$—fixing N lets them focus on a program where the outer **while** loop is fully unrolled. Martingales are also used by [15] for analyzing probabilistic termination. Finally, there are approaches involving symbolic execution; [39] use a mix of static and dynamic analysis to check probabilistic programs from the approximate computing literature.

10 Conclusion and Perspectives

We introduced an expressive program logic for probabilistic programs, and showed that assertion-based systems are suited for practical verification of probabilistic programs. Owing to their richer assertions, program logics are a more suitable foundation for specialized reasoning principles than expectation-based systems. As evidence, our program logic can be smoothly extended with custom reasoning for probabilistic independence and union bounds. Future work includes proving better accuracy bounds for differentially private algorithms, and exploring further integration of Ellora into EasyCrypt.

Acknowledgments. We thank the reviewers for their helpful comments. This work benefited from discussions with Dexter Kozen, Annabelle McIver, and Carroll Morgan. This work was partially supported by ERC Grant #679127, and NSF grant 1718220.

References

1. Baier, C.: Probabilistic model checking. In: Dependable Software Systems Engineering, NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 45, pp. 1–23. IOS Press (2016), https://doi.org/10.3233/978-1-61499-627-9-1
2. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.-Y.: EasyCrypt: A tutorial. In: Aldini, A., Lopez, J., Martinelli, F. (eds.) FOSAD 2012-2013. LNCS, vol. 8604, pp. 146–166. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10082-1_6
3. Barthe, G., Espitau, T., Ferrer Fioriti, L.M., Hsu, J.: Synthesizing probabilistic invariants via Doob's decomposition. In: International Conference on Computer Aided Verification (CAV), Toronto, Ontario (2016). https://arxiv.org/abs/1605.02765
4. Barthe, G., Gaboardi, M., Grégoire, B., Hsu, J., Strub, P.Y.: A program logic for union bounds. In: International Colloquium on Automata, Languages and Programming (ICALP), Rome, Italy (2016). http://arxiv.org/abs/1602.05681
5. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-aided security proofs for the working cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22792-9_5
6. Bellare, M., Rogaway, P.: The security of triple encryption and a framework for code-based game-playing proofs. In: IACR International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT), Saint Petersburg, Russia, pp. 409–426 (2006). https://doi.org/10.1007/11761679_25
7. Bizjak, A., Birkedal, L.: Step-indexed logical relations for probability. In: Pitts, A. (ed.) FoSSaCS 2015. LNCS, vol. 9034, pp. 279–294. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46678-0_18
8. Chadha, R., Cruz-Filipe, L., Mateus, P., Sernadas, A.: Reasoning about probabilistic sequential programs. Theoretical Computer Science **379**(1–2), 142–165 (2007)
9. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 511–526. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_34
10. Chakarov, A., Sankaranarayanan, S.: Expectation invariants for probabilistic program loops as fixed points. In: Müller-Olm, M., Seidl, H. (eds.) SAS 2014. LNCS, vol. 8723, pp. 85–100. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10936-7_6
11. Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Saint Petersburg, Florida, pp. 327–342 (2016). https://doi.org/10.1145/2837614.2837639
12. Cousot, P., Monerau, M.: Probabilistic abstract interpretation. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 169–193. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_9
13. Crubillé, R., Dal Lago, U.: On probabilistic applicative bisimulation and call-by-value λ-calculi. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 209–228. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_12

14. Dal Lago, U., Sangiorgi, D., Alberti, M.: On coinductive equivalences for higher-order probabilistic functional programs. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, California, pp. 297–308 (2014). https://arxiv.org/abs/1311.1722
15. Fioriti, L.M.F., Hermanns, H.: Probabilistic termination: Soundness, completeness, and compositionality. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Mumbai, India, pp. 489–501 (2015)
16. Gretz, F., Katoen, J.P., McIver, A.: Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. Perform. Eval. **73**, 110–132 (2014)
17. Hart, S., Sharir, M., Pnueli, A.: Termination of probabilistic concurrent programs. ACM Trans. Program. Lang. Syst. **5**(3), 356–380 (1983)
18. den Hartog, J.: Probabilistic extensions of semantical models. Ph.D. thesis, Vrije Universiteit Amsterdam (2002)
19. Hurd, J.: Formal verification of probabilistic algorithms. Technical report, UCAM-CL-TR-566, University of Cambridge, Computer Laboratory (2003)
20. Hurd, J.: Verification of the Miller-Rabin probabilistic primality test. J. Log. Algebr. Program. **56**(1–2), 3–21 (2003). https://doi.org/10.1016/S1567-8326(02)00065-6
21. Hurd, J., McIver, A., Morgan, C.: Probabilistic guarded commands mechanized in HOL. Theor. Comput. Sci. **346**(1), 96–112 (2005)
22. Impagliazzo, R., Rudich, S.: Limits on the provable consequences of one-way permutations. In: ACM SIGACT Symposium on Theory of Computing (STOC), Seattle, Washington, pp. 44–61 (1989). https://doi.org/10.1145/73007.73012
23. Kaminski, B.L., Katoen, J.-P., Matheja, C.: Inferring covariances for probabilistic programs. In: Agha, G., Van Houdt, B. (eds.) QEST 2016. LNCS, vol. 9826, pp. 191–206. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43425-4_14
24. Kaminski, B., Katoen, J.P., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run-times of probabilistic programs. In: European Symposium on Programming (ESOP), Eindhoven, The Netherlands, January 2016
25. Katoen, J.P.: The probabilistic model-checking landscape. In: IEEE Symposium on Logic in Computer Science (LICS), New York (2016)
26. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, pp. 482–491 (2003). https://doi.org/10.1109/SFCS.2003.1238221
27. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. **22**, 328–350 (1981). https://www.sciencedirect.com/science/article/pii/0022000081900362
28. Kozen, D.: A probabilistic PDL. J. Comput. Syst. Sci. **30**(2), 162–178 (1985)
29. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
30. McIver, A., Morgan, C.: Abstraction, refinement, and proof for probabilistic systems. Monographs in Computer Science. Springer, New York (2005)
31. McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.P.: A new rule for almost-certain termination. In: Proceedings of the ACM on Programming Languages 1(POPL) (2018). https://arxiv.org/abs/1612.01091, appeared at ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Los Angeles, California

32. Monniaux, D.: Abstract interpretation of probabilistic semantics. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 322–339. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-45099-3_17
33. Morgan, C.: Proof rules for probabilistic loops. In: BCS-FACS Conference on Refinement, Bath, England (1996)
34. Morgan, C., McIver, A., Seidel, K.: Probabilistic predicate transformers. ACM Trans. Program. Lang. Syst. **18**(3), 325–353 (1996)
35. Olmedo, F., Kaminski, B.L., Katoen, J.P., Matheja, C.: Reasoning about recursive probabilistic programs. In: IEEE Symposium on Logic in Computer Science (LICS), New York, pp. 672–681 (2016)
36. Pearl, J., Paz, A.: Graphoids: graph-based logic for reasoning about relevance relations. In: ECAI, pp. 357–363 (1986)
37. Ramshaw, L.H.: Formalizing the Analysis of Algorithms. Ph.D. thesis, Computer Science (1979)
38. Rand, R., Zdancewic, S.: VPHL: a verified partial-correctness logic for probabilistic programs. In: Conference on the Mathematical Foundations of Programming Semantics (MFPS), Nijmegen, The Netherlands (2015)
39. Sampson, A., Panchekha, P., Mytkowicz, T., McKinley, K.S., Grossman, D., Ceze, L.: Expressing and verifying probabilistic assertions. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Edinburgh, Scotland, p. 14 (2014)
40. Sharir, M., Pnueli, A., Hart, S.: Verification of probabilistic programs. SIAM J. Comput. **13**(2), 292–314 (1984)
41. Valiant, L.G.: A scheme for fast parallel communication. SIAM J. Comput. **11**(2), 350–361 (1982)
42. Valiant, L.G., Brebner, G.J.: Universal schemes for parallel communication. In: ACM SIGACT Symposium on Theory of Computing (STOC), Milwaukee, Wisconsin, pp. 263–277 (1981). https://doi.org/10.1145/800076.802479

Fine-Grained Semantics for Probabilistic Programs

Benjamin Bichsel(✉), Timon Gehr, and Martin Vechev

ETH Zürich, Zürich, Switzerland
{benjamin.bichsel,timon.gehr,martin.vechev}@inf.ethz.ch

Abstract. Probabilistic programming is an emerging technique for modeling processes involving uncertainty. Thus, it is important to ensure these programs are assigned precise formal semantics that also cleanly handle typical exceptions such as non-termination or division by zero. However, existing semantics of probabilistic programs do not fully accommodate different exceptions and their interaction, often ignoring some or conflating multiple ones into a single exception state, making it impossible to distinguish exceptions or to study their interaction.

In this paper, we provide an expressive probabilistic programming language together with a fine-grained measure-theoretic denotational semantics that handles and distinguishes non-termination, observation failures and error states. We then investigate the properties of this semantics, focusing on the interaction of different kinds of exceptions. Our work helps to better understand the intricacies of probabilistic programs and ensures their behavior matches the intended semantics.

1 Introduction

A probabilistic programming language allows probabilistic models to be specified independently of the particular inference algorithms that make predictions using the model. Probabilistic programs are formed using standard language primitives as well as constructs for drawing random values and conditioning. The overall approach is general and applicable to many different settings (e.g., building cognitive models). In recent years, the interest in probabilistic programming systems has grown rapidly with various languages and probabilistic inference algorithms (ranging from approximate to exact). Examples include [10,11,13,14,25–27,29,36]; for a recent survey, please see [15]. An important branch of recent probabilistic programming research is concerned with providing a suitable semantics for these programs enabling one to formally reason about the program's behaviors [2–4,33–35].

Often, probabilistic programs require access to primitives that may result in unwanted behavior. For example, the standard deviation σ of a Gaussian distribution must be positive (sampling from a Gaussian distribution with negative standard deviation should result in an error). If a program samples from a Gaussian distribution with a non-constant standard deviation, it is in general

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 145–185, 2018.
https://doi.org/10.1007/978-3-319-89884-1_6

undecidable if that standard deviation is guaranteed to be positive. A similar situation occurs for while loops: except in some trivial cases, it is hard to decide if a program terminates with probability one (even harder than checking termination of deterministic programs [20]). However, general while loops are important for many probabilistic programs. As an example, a Markov Chain Monte Carlo sampler is essentially a special probabilistic program, which in practice requires a non-trivial stopping criterion (see e.g. [6] for such a stopping criterion). In addition to offering primitives that may result in such unwanted behavior, many probabilistic programming languages also provide an **observe** primitive that intuitively allows to filter out executions violating some constraint.

Motivation. Measure-theoretic denotational semantics for probabilistic programs is desirable as it enables reasoning about probabilistic programs within the rigorous and general framework of measure theory. While existing research has made substantial progress towards a rigorous semantic foundation of probabilistic programming, existing denotational semantics based on measure theory usually conflate failing **observe** statements (i.e., conditioning), error states and non-termination, often modeling at least some of these as missing weight in a sub-probability measure (we show why this is practically problematic in later examples). This means that even semantically, it is impossible to distinguish these types of exceptions[1]. However, distinguishing exceptions is essential for a solid understanding of probabilistic programs: it is insufficient if the semantics of a probabilistic programming language can only express that *something* went wrong during the execution of the program, lacking the capability to distinguish for example non-termination and errors. Concretely, programmers often want to avoid non-termination and assertion failure, while observation failure is acceptable (or even desirable). When a program runs into an exception, the programmer should be able determine the type of exception, from the semantics.

This Work. This paper presents a clean denotational semantics for a Turing complete first-order probabilistic programming language that supports mixing continuous and discrete distributions, arrays, observations, partial functions and loops. This semantics distinguishes observation failures, error states and non-termination by tracking them as explicit program states. Our semantics allows for fine-grained reasoning, such as determining the termination probability of a probabilistic program making observations from a sequence of concrete values.

In addition, we explain the consequences of our treatment of exceptions by providing interesting examples and properties of our semantics, such as commutativity in the absence of exceptions, or associativity regardless of the presence of exceptions. We also investigate the interaction between exceptions and the **score** primitive, concluding in particular that the probability of non-termination cannot be defined in this case. **score** intuitively allows to increase or decrease the probability of specific runs of a program (for more details, see Sect. 5.3).

[1] In this paper, we refer to errors, non-termination and observation failures collectively as *exceptions*. For example, a division by zero is an error (and hence and exception), while non-termination is an exception but not an error.

2 Overview

In this section we demonstrate several important features of our probabilistic programming language (PPL) using examples, followed by a discussion involving different kinds of exception interactions.

2.1 Features of Probabilistic Programs

In the following, we informally discuss the most important features of our PPL.

Discrete and Continuous Primitive Distributions. Listing 1 illustrates a simple Gaussian mixture model (the figure only shows the function body). Depending on the outcome of a fair coin flip x (resulting in 0 or 1), y is sampled from a Gaussian distribution with mean 0 or mean 2 (and standard deviation 1). Note that in our PPL, we represent **gauss**$(\cdot,\cdot)$ by the more general construct **sampleFrom**$_f(\cdot,\cdot)$, with $f : \mathbb{R} \times [0,\infty) \to \mathbb{R} \to \mathbb{R}$ being the probability density function of the Gaussian distribution $f(\mu,\sigma)(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$.

```
y:=0;
if flip(½) {
  y=gauss(0,1);
}else{
  y=gauss(2,1);
}
return y;
```

Listing 1. Simple Gaussian mixture

Conditioning. Listing 2 samples two independent values from the uniform distribution on the interval $[0,1]$ and conditions the possible values of x and y on the observation $x+y>1$ before returning x. Intuitively, the first two lines express a-priori knowledge about the uncertain values of x and y. Then, a measurement determines that $x+y$ is greater than 1. We combine this new information with the existing knowledge. Because $x+y>1$ is more likely for larger values of x, the return value has larger weight on larger values. Formally, our semantics handles **observe** by introducing an extra program state for observation failure $\lightning$. Hence, the probability distribution after the third line of Listing 2 will put weight $\frac{1}{2}$ on $\lightning$ and weight $\frac{1}{2}$ on those x and y satisfying $x+y>1$.

```
x:=uniform(0,1);
y:=uniform(0,1);
observe(x+y>1);
return x;
```

Listing 2. Conditioning on a continuous distribution

In practice, one will usually condition the output distribution on there being no observation failure ($\lightning$). For discrete distributions, this amounts to computing:

$$Pr[X = x \mid X \neq \lightning] = \frac{Pr[X = x \wedge X \neq \lightning]}{Pr[X \neq \lightning]} = \frac{Pr[X = x]}{1 - Pr[X = \lightning]}$$

where x is the outcome of the program (a value, non-termination or an error) and $Pr[X = x]$ is the probability that the program results in x. Of course, this conditioning only works when the probability of $\lightning$ is not 1. Note that tracking the probability of $\lightning$ has the practical benefit of rendering the (often expensive) marginalization $Pr[X \neq \lightning] = \sum_{x \neq \lightning} Pr[X = x]$ unnecessary.

Other semantics often use sub-probability measures to express failed observations [4,34,35]. These semantics would say that Listing 2 results in a return value between 0 and 1 with probability $\frac{1}{2}$ (and infer that the missing weight of $\frac{1}{2}$ is due to failed observations). We believe one should improve upon this approach as the semantics only implicitly states that the program sometimes fails an observation. Further, this strategy only allows tracking a single kind of exception (in this case, failed observations). This has led some works to conflate observation failure and non-termination [18,34]. We believe there is an important distinction between the two: observation failure means that the program behavior is inconsistent with observed facts, non-termination means that the program did not return a result.

Listing 3 illustrates that it is not possible to condition parts of the program on there being no observation failure. In Listing 3, conditioning the first branch $x := 0;$ **observe**(**flip**($\frac{1}{2}$)) on there being no observation failure yields $Pr[x = 0] = 1$, rendering the observation irrelevant. The same situation arises for the second branch. Hence, conditioning the two branches in isolation yields $Pr[x = 0] = \frac{1}{2}$ instead of $Pr[x = 0] = \frac{2}{3}$.

```
if flip(1/2) {
  x:=0;
  observe(flip(1/2));
}else{
  x:=1;
  observe(flip(1/4));
}
```

Listing 3. The need for tracking $\lightning$

Loops. Listing 4 shows a probabilistic program with a while loop. It samples from the **geometric**($\frac{1}{2}$) distribution, which counts the number of failures (**flip** returns 0) until the first success occurs (**flip** returns 1). This program terminates with probability 1, but it is of course possible that a probabilistic program fails to terminate with positive probability. Listing 5 demonstrates this possibility.

```
n:=0;
while !flip(1/2) {
  n=n+1;
}
return n;
```

Listing 4. Geometric distribution

Listing 5 modifies x until either $x = 0$ or $x = 10$. In each iteration, x is either increased or decreased, each with probability $\frac{1}{2}$. If x reaches 0, the loop terminates. If x reaches 10, the loop never terminates. By symmetry, both termination and non-termination are equally likely. Hence, the program either returns 0 or does not terminate, each with probability $\frac{1}{2}$.

```
x := 5;
while x>0 {
  if x<10 {
    x+=2*flip(1/2)-1;
  }
}
return x;
```

Listing 5. Program that may not terminate

Other semantics often use sub-probability measures to express non-termination [4,23]. Thus, these semantics would say that Listing 5 results in 0 with probability $\frac{1}{2}$ (and nothing else). We propose to track the probability of non-termination explicitly by an additional state $\circlearrowleft$, just as we track the probability of observation failure ($\lightning$).

Partial Functions. Many functions that are practically useful are only partial (meaning they are not defined for some inputs). Examples include **uniform**(a, b) (undefined for $b < a$) and $\sqrt{x}$ (undefined for $x < 0$). Listing 6 shows an example program using $\sqrt{x}$. Usually, semantics do not explicitly address partial functions [23,24,28,33] or use

```
x:=uniform(-1,1);
x=√x;
return x;
```

Listing 6. Using partial functions

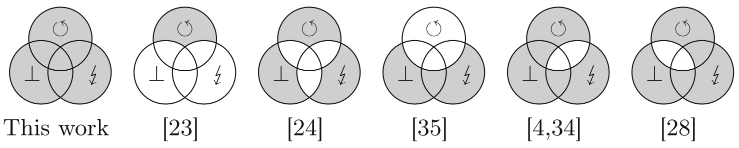


Fig. 1. Visual comparison of the exception handling capabilities of different semantics. For example, ↻ is filled in [34] because its semantics can handle non-termination. However, the intersection between ↻ and ↯ is not filled because [34] cannot distinguish non-termination from observation failure.

partial functions without dealing with failure (e.g. [19] use **Bernoulli**(p) without stating what happens if $p \notin [0, 1]$). Most of these languages could use a sub-probability distribution that misses weight in the presence of errors (in these languages, this results in conflating errors with non-termination and observation failures).

We introduce a third exception state $\bot$ that can be produced when partial functions are evaluated outside of their domain. Thus, Listing 6 results in $\bot$ with probability $\frac{1}{2}$ and returns a value from $[0, 1]$ with probability $\frac{1}{2}$ (larger values are more likely). Some previous work uses an error state to capture failing computations, but does not propagate this failure implicitly [34,35]. In particular, if an early expression in a long program may fail evaluating $\sqrt{-4}$, every expression in the program that depends on this failing computation has to check whether an exception has occurred. While it may seem possible to skip the rest of the function in case of a failing computation (by applying the pattern **if** $(x = \bot)$ $\{$**return** $\bot\}$ **else** $\{$rest of function$\}$), this is non-modular and does not address the result of the function being used in other parts of a program.

Although our semantics treat $\bot$ and ↯ similarly, there is an important distinction between the two: $\bot$ means the program terminated due to an error, while ↯ means that according to observed evidence, the program did not actually run.

2.2 Interaction of Exception States

Next, we illustrate the interaction of different exception states. We explain how our semantics handles these interactions when compared to existing semantics. Fig. 1 gives an overview of which existing semantics can handle which (interactions of) exceptions. We note that our semantics could easily distinguish more kinds of exceptions, such as division by zero or out of bounds accesses to arrays.

Non-termination and Observation Failure. Listing 7 shows a program that has been investigated in [22]. Based on the observations, it only admits a single behavior, namely always sampling $x = 0$ in the third line. This behavior results in non-termination, but it occurs with probability 0. Hence, the program fails an observation (ending up in state ↯) with probability 1. If we try to

```
x:=0;
while x=0 {
  x=flip(½);
  observe(x=0);
}
```

Listing 7. Mixing loops and observations

condition on not failing any observation (by rescaling appropriately), this results in a division by 0, because the probability of not failing any observation is 0.

The semantics of Listing 7 thus only has weight on $\lightning$, and does not allow conditioning on not failing any observation. This is also the solution that [22] proposes, but in our case, we can formally back up this claim with our semantics.

Other languages handle both non-termination and observation failure by subprobability distributions, which makes it impossible to conclude that the missing weight is due to observation failure (and not due to non-termination) [4,24,34]. The semantics in [28] cannot directly express that the missing weight is due to observation failure (rather, the semantics are undefined due to a division by zero). However, the semantics enables a careful reader to determine that the missing weight is due to observation failure (by investigating the conditional weakest precondition and the conditional weakest liberal precondition). Some other languages can express neither while loops nor observations [23,33,35].

Assertions and Non-termination. For some programs, it is useful to check assumptions explicitly. For example, the implementation of the factorial function in Listing 8 explicitly checks whether x is a valid argument to the factorial function. If $x \notin \mathbb{N}$, the program should run into an error (i.e. only have weight on $\bot$). If $x \in \mathbb{N}$, the program should return $x!$ (i.e. only have weight on $x!$). This example illustrates that earlier exceptions (like failing an assertion) should *bypass* later exceptions (like non-termination, which occurs for $x \notin \mathbb{N}$ if the programmer forgets the first two assertions). This is not surprising, given that this is also the semantics of exceptions in most deterministic languages. Most existing semantics either cannot express Listing 8 ([23,34] have no assertions, [35] has no iteration) or cannot distinguish failing an assertion from non-termination [24,28,33]. The consequence of the latter is that removing the first two assertions from Listing 8 does not affect the semantics. Handling assertion failure by sum types (as e.g. in [34]) could be a solution, but would force the programmer to deal with assertion failure explicitly. Only the semantics in [4] has the expressiveness to implicitly handle assertion errors in Listing 8 without conflating those errors with non-termination.

```
assert(x≥0);
assert(x=⌊x⌋);
fac:=1;
while x≠0 {
  fac=fac*x;
  x=x-1;
}
return fac;
```

Listing 8. Explicitly checking assumptions

Listing 9 shows a different interaction between non-termination and failing assertions. Here, even though the loop condition is always true, the first iteration of the loop will run into an exception. Thus, Listing 9 results in $\bot$ with probability 1. Again, this behavior should not be surprising given the behavior of deterministic languages. For Listing 9, conflating errors with non-termination means the program semantics cannot express that the missing weight is due to an error and not due to non-termination.

```
x:=0;
while 1 {
  x=x/x;
}
```

Listing 9. Guaranteed failure

```
observe(flip(1/2));
assert(flip(1/2));
```

Listing 10. Observation or assertion failure

Observation Failure and Assertion Failure. In our PPL, earlier exceptions bypass later exceptions, as illustrated in Listing 8. However, because we are operating in a probabilistic language, exceptions can occur probabilistically. Listing 10 shows a program that may run into an observation failure, or into an assertion failure, or neither. If it runs into an observation failure (with probability $\frac{1}{2}$), it bypasses the rest of the program, resulting in $\lightning$ with probability $\frac{1}{2}$ and in $\bot$ with probability $\frac{1}{4}$. Conditioning on the absence of observation failures, the probability of $\bot$ is $\frac{1}{2}$.

An important observation is that reordering the two statements of Listing 10 will result in a different behavior. This is the case, even though there is no obvious data-flow between the two statements. This is in sharp contrast to the semantics in [34], which guarantee (in the absence of exceptions) that only data flow is relevant and that expressions can be reordered. Our semantics illustrate that even if there is no explicit data-dependency, some seemingly obvious properties (like commutativity) may not hold in the presence of exceptions. Some languages either cannot express Listing 10 ([23,33] lack observations), cannot distinguish observation failure from assertion failure [24] or cannot handle exceptions implicitly [34,35].

Summary. In this section, we showed examples of probabilistic programs that exhibit non-termination, observation failures and errors. Then, we provided examples that show how these exceptions can interact, and explained how existing semantics handle these interactions.

3 Preliminaries

In this section, we provide the necessary theory. Most of the material is standard, however, our treatment of exception states is interesting and important for providing semantics to probabilistic programs in the presence of exceptions. All key lemmas (together with additional definitions and examples) are proven in Appendix A.

Natural Numbers, $[n]$, Iverson Brackets, Restriction of Functions. We include 0 in the natural numbers, so that $\mathbb{N} := \{0, 1, \dots\}$. For $n \in \mathbb{N}$, $[n] := \{1, \dots, n\}$. The *Iverson brackets* $[\cdot]$ are defined by $[b] = 1$ if b is true and $[b] = 0$ if b is false. A particular application of the Iverson brackets is to characterize the indicator function of a specific set S by $[x \in S]$. For a function $f \colon X \to Y$ and a subset of the domain $S \subseteq X$, f restricted to S is denoted by $f_{|S} \colon S \to Y$.

Set of Variables, Generating Tuples, Preservation of Properties, Singleton Set. Let Vars be a set of admissible variable names. We refer to the elements of Vars by x, y, z and x_i, y_i, z_i, v_i, w_i, for $i \in \mathbb{N}$. For $v \in A$ and $n \in \mathbb{N}$, $v!n := (v, \dots, v) \in A^n$ denotes the tuple containing n copies of v. A function $f \colon A^n \to A$ *preserves a property* if whenever $a_1, \dots, a_n \in A$ have that property, $f(a_1, \dots, a_n) \in A$ has that property. Let $\mathbb{1}$ denote the set which only contains the empty tuple $()$, i.e. $\mathbb{1} := \{()\}$. For sets of tuples $S \subseteq \prod_{i=1}^{n} A_i$, there is an isomorphism $S \times \mathbb{1} \simeq \mathbb{1} \times S \simeq S$. This isomorphism is intuitive and we sometimes silently apply it.

Exception States, Lifting Functions to Exception States. We allow the extension of sets with some symbols that stand for the occurrence of special events in a program. This is important because it allows us to capture the event that a given program runs into specific exceptions. Let $\mathcal{X} := \{\bot, \lightning, \circlearrowleft\}$ be a (countable) set of exception states. We denote by $\overline{A} := A \cup \mathcal{X}$ the set A extended with $\mathcal{X}$ (we require that $A \cap \mathcal{X} = \emptyset$). Intuitively, $\bot$ corresponds to assertion failures, $\lightning$ corresponds to observation failures and $\circlearrowleft$ corresponds to non-termination. For a function $f \colon A \to B$, *f lifted to exception states*, denoted by $\overline{f} \colon \overline{A} \to \overline{B}$ is defined by $\overline{f}(a) = a$ if $a \in \mathcal{X}$ and $\overline{f}(a) = f(a)$ if $a \notin \mathcal{X}$. For a function $f \colon \prod_{i=1}^{n} A_i \to B$, *$f$ lifted to exception states*, denoted by $\overline{f} \colon \prod_{i=1}^{n} \overline{A_i} \to \overline{B}$, propagates the first exception in its arguments, or evaluates f if none of its arguments are exceptions. Formally, it is defined by $\overline{f}(a_1, \dots, a_n) = a_1$ if $a_1 \in \mathcal{X}$, $\overline{f}(a_1, \dots, a_n) = a_2$ if $a_1 \notin \mathcal{X}$ and $a_2 \in \mathcal{X}$, and so on. Only if $a_1, \dots, a_n \notin \mathcal{X}$, we have $\overline{f}(a_1, \dots, a_n) = f(a_1, \dots, a_n)$. Thus, $\overline{f}(\circlearrowleft, a, \bot) = \circlearrowleft$. In particular, we write $\overline{(a, b)}$ for lifting the tupling function, resulting in for example $\overline{(\lightning, \circlearrowleft)} = \lightning$. To remove notation clutter, we do not distinguish the two different liftings $\overline{f} \colon \overline{A} \to \overline{B}$ and $\overline{f} \colon \prod_{i=1}^{n} \overline{A_i} \to \overline{B}$ notationally. Whenever we write $\overline{f}$, it will be clear from the context which lifting we mean. We write $S \overline{\times} T$ for $\{\overline{(s, t)} \mid s \in S, t \in T\}$.

Records. A *record* is a special type of tuple indexed by variable names. For sets $(S_i)_{i \in [n]}$, a record $r \in \prod_{i=1}^{n} (x_i \colon S_i)$ has the form $r = \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$, where $v_i \in S_i$, with the convenient shorthand $r = \{x_i \mapsto v_i\}_{i \in [n]}$. We can access the elements of a record by their name: $r[x_i] = v_i$.

In what follows, we provide the measure theoretic background necessary to express our semantics.

σ-algebra, Measurable Set, σ-algebra Generated by a Set, Measurable Space, Measurable Functions. Let A be some set. A set $\Sigma_A \subseteq \mathcal{P}(A)$ is called a *σ-algebra on A* if it satisfies three conditions: $A \in \Sigma_A$, Σ_A is closed under complements ($S \in \Sigma_A$ implies $A \backslash S \in \Sigma_A$) and Σ_A is closed under countable unions (for any collection $\{S_i\}_{i \in \mathbb{N}}$ with $S_i \in \Sigma_A$, we have $\bigcup_{i \in \mathbb{N}} S_i \in \Sigma_A$). The elements of Σ_A are called *measurable sets*. For any set A, a trivial σ-algebra on A is its power set $\mathcal{P}(A)$. Unfortunately, the power set often contains sets that do not behave well. To come up with a σ-algebra on A whose sets do behave well, we often start with a set $S \subseteq \mathcal{P}(A)$ that is not a σ-algebra and extend it until we get a σ-algebra. For this purpose, let A be some set and $S \subseteq \mathcal{P}(A)$ a collection of subsets of A. The *σ-algebra generated by S* denoted by $\sigma(S)$ is the smallest σ-algebra that contains S. Formally, $\sigma(S)$ is the intersection of all σ-algebras on A containing S. For a set A and a σ-algebra Σ_A on A, (A, Σ_A) is called a *measurable space*. We often leave Σ_A implicit; whenever it is not mentioned explicitly, it is clear from the context. Table 1 provides the implicit σ-algebras for some common sets. As an example, some elements of $\Sigma_{\overline{\mathbb{R}}}$ include $[0, 1] \cup \{\bot\}$ and $\{1, 3, \pi\}$. For measurable spaces (A, Σ_A) and (B, Σ_B), a function $f \colon A \to B$ is called *measurable*,

Table 1. Implicit σ-algebras on common sets, for measurable spaces (A, Σ_A), (A_i, Σ_{A_i})

Set	σ-algebra on this set
$\mathbb{R}$	$\Sigma_{\mathbb{R}} = \mathcal{B} := \sigma(\{[a,b] \subseteq \mathbb{R} \mid a \leq b, a \in \mathbb{R}, b \in \mathbb{R}\})$, the Borel σ-algebra on $\mathbb{R}$ generated by all intervals
S for $S \in \mathcal{B}$	$\Sigma_S = \{T \in \mathcal{B} \mid T \subseteq S\}$
$\prod_{i=1}^n A_i$	$\Sigma_{\prod_{i=1}^n A_i} = \sigma\left(\{\prod_{i=1}^n S_i \mid S_i \in \Sigma_{A_i}\}\right)$
$\prod_{i=1}^n (x_i : A_i)$	$\Sigma_{\prod_{i=1}^n (x_i : A_i)} = \sigma\left(\{\prod_{i=1}^n (x_i : S_i) \mid S_i \in \Sigma_{A_i}\}\right)$
$\overline{A}$	$\Sigma_{\overline{A}} = \{S \cup S' \mid S \in \Sigma_A, S' \in \mathcal{P}(\mathcal{X})\}$

if $\forall S \in \Sigma_B \colon f^{-1}(S) \in \Sigma_A$. Here, $f^{-1}(S) := \{a \in A \colon f(a) \in S\}$. If one is familiar with the notion of Lebesgue measurable functions, note that our definition does not include all Lebesgue measurable functions. As a motivation to why we need measurable functions, consider the following scenario. We know the distribution of some variable x, and want to know the distribution of $y = f(x)$. To figure out how likely it is that $y \in S$ for a measurable set S, we can determine how likely it is that $x \in f^{-1}(S)$, because $f^{-1}(S)$ is guaranteed to be a measurable set.

Measures, Examples of Measures. For a measurable space (A, Σ_A), a function $\mu \colon \Sigma_A \to [0, \infty]$ is called a *measure on A* if it satisfies two properties: null empty set ($\mu(\emptyset) = 0$) and countable additivity (for any countable collection $\{S_i\}_{i \in \mathcal{I}}$ of pairwise disjoint sets $S_i \in \Sigma_A$, we have $\mu\left(\bigcup_{i \in \mathcal{I}} S_i\right) = \sum_{i \in \mathcal{I}} \mu(S_i)$). Measures allow us to quantify the probability that a certain result lies in a measurable set. For example, $\mu([1, 2])$ can be interpreted as the probability that the outcome of a process is between 1 and 2.

The *Lebesgue measure* $\lambda \colon \mathcal{B} \to [0, \infty]$ is the (unique) measure that satisfies $\lambda([a, b]) = b - a$ for all $a, b \in \mathbb{R}$ with $a \leq b$. The *zero measure* $\mathbf{0} \colon \Sigma_A \to [0, \infty]$ is defined by $\mathbf{0}(S) = 0$ for all $S \in \Sigma_A$. For a measurable space (A, Σ_A) and some $a \in A$, the *Dirac measure* $\delta_a \colon \Sigma_A \to [0, \infty]$ is defined by $\delta_a(S) = [a \in S]$.

Unfortunately, there are measures that do not satisfy some important properties (for example, they may not satisfy Fubini's theorem, which we discuss later on). The usual way to deal with this is to restrict our attention to σ-finite measures, which are well-known and were studied in great detail. However, σ-finite measures are too restrictive for our purposes. In particular, the s-finite kernels that we introduce later on can induce measures that are not σ-finite. This is why in the following, we work with s-finite measures. Table 2 gives an overview of the different kinds of measures that are important for understanding our work. The expression $1/2 \cdot \delta_1$ stands for the pointwise multiplication of the measure δ_1 by 1/2: $1/2 \cdot \delta_1 = \lambda S. 1/2 \cdot \delta_1(S)$. Here, the λ refers to λ-abstraction and not to the Lebesgue measure. To distinguish the two λs, we always write "$\lambda x.$" (with a dot) when we refer to λ-abstraction. For more details on the definitions and for proofs about the provided examples, see Appendix A.1.

Table 2. Definition and comparison of different measures $\mu\colon \Sigma_A \to [0,\infty]$ on measurable spaces (A, Σ_A). Reading the table top-down, we get from the most restrictive definition to the most permissive definition. For example, any sub-probability measure is also a σ-finite measure. We also provide an example for each type of measure that is not an example of the more restrictive type of measure. For example, the Lebesgue measure λ is σ-finite but not s-finite.

Type of measure	Characterization	Examples
Probability measure	μ is a measure and $\mu(A) = 1$	$\mu = \delta_1$
Sub-probability measure	μ is a measure and $\mu(A) \leq 1$	$\mu = \mathbf{0}$ or $\mu = 1/2 \cdot \delta_1$
σ-finite measure	μ is a measure and $A = \bigcup_{i\in\mathbb{N}} A_i$ for $A_i \in \Sigma_A$ with $\mu(A_i) < \infty$	$\mu = \lambda$
s-finite measure	$\mu = \sum_{i\in\mathbb{N}} \mu_i$ for sub-probability measures μ_i	$\mu(S) = \begin{cases} 0 & \lambda(S) = 0 \\ \infty & \lambda(S) > 0 \end{cases}$
Measure	$\mu(\emptyset) = 0$, countable additivity	$\mu(S) = \begin{cases} \lvert S\rvert & S \text{ finite} \\ \infty & \text{otherwise} \end{cases}$

Product of Measures, Product of Measures in the Presence of Exception States. For s-finite measures $\mu\colon \Sigma_A \to [0,\infty]$ and $\mu'\colon \Sigma_B \to [0,\infty]$, we denote the *product of measures* by $\mu \times \mu'\colon \Sigma_{A\times B} \to [0,\infty]$, and define it by

$$(\mu \times \mu')(S) = \int_{a\in A} \int_{b\in B} [(a,b) \in S]\mu'(db)\mu(da)$$

For s-finite measures $\mu\colon \Sigma_{\overline{A}} \to [0,\infty]$ and $\mu'\colon \Sigma_{\overline{B}} \to [0,\infty]$, we denote the *lifted product of measures* by $\mu \overline{\times} \mu'\colon \Sigma_{\overline{A\times B}} \to [0,\infty]$ and define it using the lifted tupling function: $(\mu\overline{\times}\mu')(S) = \int_{a\in\overline{A}} \int_{b\in\overline{B}} [\overline{(a,b)} \in S]\mu'(db)\mu(da)$. While the product of measures $\mu \times \mu'$ is well known for combining two measures to a joint measure, the concept of a lifted product of measures $\mu\overline{\times}\mu'$ is required to do the same for combining measures that have weight on exception states. Because the formal semantics of our probabilistic programming language makes use of exception states, we always use $\overline{\times}$ to combine measures, appropriately handling exception states implicitly.

Lemma 1. *For measures $\mu\colon \Sigma_A \to [0,\infty]$, $\mu'\colon \Sigma_B \to [0,\infty]$, let $S \in \Sigma_A$ and $T \in \Sigma_B$. Then, $(\mu \times \mu')(S \times T) = \mu(S) \cdot \mu'(T)$.*

For $\mu\colon \Sigma_{\overline{A}} \to [0,\infty]$, $\mu'\colon \Sigma_{\overline{B}} \to [0,\infty]$ and $S \in \Sigma_{\overline{A}}$, $T \in \Sigma_{\overline{B}}$, in general we have $(\mu\overline{\times}\mu')(S \times T) \neq \mu(S) \cdot \mu'(T)$, due to interactions of exception states.

Lemma 2. *$\times$ and $\overline{\times}$ for s-finite measures are associative, left- and right-distributive and preserve (sub-)probability and s-finite measures.*

Lebesgue Integrals, Fubini's Theorem for s-finite Measures. Our definition of the Lebesgue integral is based on [31]. It allows integrating functions that sometimes evaluate to ∞, and Lebesgue integrals evaluating to ∞.

Here, (A, Σ_A) and (B, Σ_B) are measurable spaces and $\mu\colon \Sigma_A \to [0, \infty]$ and $\mu'\colon \Sigma_B \to [0, \infty]$ are measures on A and B, respectively. Also, $E \in \Sigma_A$ and $F \in \Sigma_B$. Let $s\colon A \to [0, \infty)$ be a measurable function. s is a *simple function* if $s(x) = \sum_{i=1}^{n} \alpha_i [x \in A_i]$ for $A_i \in \Sigma_A$ and $\alpha_i \in \mathbb{R}$. For any simple function s, the Lebesgue integral of s over E with respect to μ, denoted by $\int_{a\in E} s(a)\mu(da)$, is defined by $\sum_{i=1}^{n} \alpha_i \cdot \mu(A_i \cap E)$, making use of the convention $0 \cdot \infty = 0$. Let $f\colon A \to [0, \infty]$ be measurable but not necessarily simple. Then, the *Lebesgue integral* of f over E with respect to μ is defined by

$$\int_{a\in E} f(a)\mu(da) := \sup \left\{ \int_{a\in E} s(a)\mu(da) \;\middle|\; s : A \to [0, \infty) \text{ is simple}, 0 \leq s \leq f \right\}$$

Here, the inequalities on functions are pointwise. Appendix A.2 lists some useful properties of the Lebesgue integral. Here, we only mention Fubini's theorem, which is important because it entails a commutativity-like property of the product of measures: $(\mu \times \mu')(S) = (\mu' \times \mu)(\mathsf{swap}(S))$, where swap switches the dimensions of S: $\mathsf{swap}(S) = \{(b, a) \mid (a, b) \in S\}$. The proof of this property is straightforward, by expanding the definition of the product of measures and applying Fubini's theorem. As we show in Sect. 5, this property is crucial for the commutativity of expressions. In the presence of exceptions, it does not hold: $(\mu \overline{\times} \mu')(S) \neq (\mu' \overline{\times} \mu)(\mathsf{swap}(S))$ in general.

Theorem 1 (Fubini's theorem). *For s-finite measures $\mu\colon \Sigma_A \to [0, \infty]$ and $\mu'\colon \Sigma_B \to [0, \infty]$ and any measurable function $f\colon A \times B \to [0, \infty]$,*

$$\int_{a\in A} \int_{b\in B} f(a, b)\mu'(db)\mu(da) = \int_{b\in B} \int_{a\in A} f(a, b)\mu(da)\mu'(db)$$

For s-finite measures $\mu\colon \Sigma_{\overline{A}} \to [0, \infty]$ and $\mu'\colon \Sigma_{\overline{B}} \to [0, \infty]$ and any measurable function $f\colon A \times B \to [0, \infty]$,

$$\int_{a\in \overline{A}} \int_{b\in \overline{B}} \overline{f}(a, b)\mu'(db)\mu(da) = \int_{b\in \overline{B}} \int_{a\in \overline{A}} \overline{f}(a, b)\mu(da)\mu'(db)$$

(Sub-)probability Kernels, s-finite Kernels, Dirac Delta, Lebesgue Kernel, Motivation for s-finite Kernels. In the following, let (A, Σ_A) and (B, Σ_B) be measurable spaces. A *(sub-)probability kernel with source A and target B* is a function $\kappa\colon A \times \Sigma_B \to [0, \infty)$ such that for all $a \in A$: $\kappa(a, \cdot)\colon \Sigma_B \to [0, \infty)$ is a (sub-)probability measure, and $\forall S \in \Sigma_B\colon \kappa(\cdot, S)\colon A \to [0, \infty)$ is measurable. $\kappa\colon A \times \Sigma_B \to [0, \infty]$ is an *s-finite kernel with source A and target B* if κ is a pointwise sum of sub-probability kernels $\kappa_i\colon A \times \Sigma_B \to [0, \infty)$, meaning $\kappa = \sum_{i\in\mathbb{N}} \kappa_i$. We denote the set of s-finite kernels with source A and target B by $A \mapsto B \subseteq A \times \Sigma_B \to [0, \infty]$. Because we only ever deal with s-finite kernels, we often refer to them simply as kernels.

We can understand the Dirac measure as a probability kernel. For a measurable space (A, Σ_A), the *Dirac delta* $\delta\colon A \mapsto A$ is defined by $\delta(a, S) = [a \in S]$. Note that for any a, $\delta(a, \cdot)\colon \Sigma_A \to [0, \infty]$ is the Dirac measure. We often write

$\delta(a)(S)$ or $\delta_a(S)$ for $\delta(a, S)$. Note that we can also interpret $\delta\colon A \mapsto A$ as an s-finite kernel from $A \mapsto B$ for $A \subseteq B$. The *Lebesgue kernel* $\lambda^*\colon A \mapsto \mathbb{R}$ is defined by $\lambda^*(a)(S) = \lambda(S)$, where λ is the Lebesgue measure. The definition of s-finite kernels is a lifting of the notion of s-finite measures. Note that for an s-finite kernel κ, $\kappa(a, \cdot)$ is an s-finite measure for all $a \in A$. In the context of probabilistic programming, s-finite kernels have been used before [34].

Working in the space of sub-probability kernels is inconvenient, because, for example, $\lambda^*\colon \mathbb{R} \mapsto \mathbb{R}$ is not a sub-probability kernel. Even though $\lambda^*(x)$ is σ-finite measure for all $x \in \mathbb{R}$, not all s-finite kernels induce σ-finite measures in this sense. As an example, $(\lambda^*;\lambda^*)(x)$ is not a σ-finite measure for any $x \in \mathbb{R}$ (see Lemma 15 in Appendix A.1). We introduce (;) shortly in Definition 1.

Working in the space of s-finite kernels is convenient because s-finite kernels have many nice properties. In particular, the set of s-finite kernels $A \mapsto B$ is the smallest set that contains all sub-probability kernels with source A and target B and is closed under countable sums.

Lifting Kernels to Exception States, Removing Weight from Exception States. For kernels $\kappa\colon A \mapsto B$ or kernels $\kappa\colon A \mapsto \overline{B}$, κ *lifted to exception states* $\overline{\kappa}\colon \overline{A} \mapsto \overline{B}$ is defined by $\overline{\kappa}(a) = \kappa(a)$ if $a \in A$ and $\overline{\kappa}(a) = \delta(a)$ if $a \notin A$. When transforming κ into $\overline{\kappa}$, we preserve (sub-)probability and s-finite kernels.

Composing kernels, composing kernels in the presence of exception states.

Definition 1. *Let* $(;)\colon (A \mapsto B) \to (B \mapsto C) \to (A \mapsto C)$ *be defined by* $(f;g)(a)(S) = \int_{b\in B} g(b)(S)\, f(a)(db)$.

Note that $f;g$ intuitively corresponds to first applying f and then g. Throughout this paper, we mostly use $>\!\!=\!\!>$ instead of (;), but we introduce (;) because it is well-known and it is instructive to show how our definition of $>\!\!=\!\!>$ relates to (;).

Lemma 3. (;) *is associative, left- and right-distributive, has neutral element*[2] δ *and preserves (sub-)probability and s-finite kernels.*

Definition 2. *Let* $(>\!\!=\!\!>)\colon (A \mapsto \overline{B}) \to (B \mapsto \overline{C}) \to (A \mapsto \overline{C})$ *be defined by* $(f >\!\!=\!\!> g)(a)(S) = \int_{b\in\overline{B}} \overline{g}(b)(S)\, f(a)(db)$.

We sometimes write $f(a) \gg\!= g$ for $(f >\!\!=\!\!> g)(a)$.

Lemma 4. *For* $f\colon A \mapsto \overline{B}$ *and* $g\colon B \mapsto \overline{C}$, $a \in A$ *and* $S \in \Sigma_{\overline{C}}$,

$$(f >\!\!=\!\!> g)(a)(S) = (f;g)(a)(S) + \sum_{x\in\mathcal{X}} \delta(x)(S) f(a)(\{x\})$$

Lemma 4 shows how $>\!\!=\!\!>$ relates to (;), by splitting $f >\!\!=\!\!> g$ into non-exceptional behavior of f (handled by (;)) and exceptional behavior of f (handled by a sum). Intuitively, if f produces an exception state $\star \in \mathcal{X}$, then g is not even evaluated. Instead, this exception is directly passed on, as indicated by $\delta(x)(S)$.

[2] δ is a neutral element of (;) if $(\delta;\kappa) = (\kappa;\delta) = \kappa$ for all kernels κ.

If $f(a)(\mathcal{X}) = 0$ for all $a \in A$, or if $S \cap \mathcal{X} = \emptyset$, then the definitions are equivalent in the sense that $(f;g)(a)(S) = (f \ggg g)(a)(S)$. The difference between $\ggg$ and (;) is the treatment of exception states produced by f. Note that technically, the target $\overline{B}$ of $f \colon A \mapsto \overline{B}$ does not match the source B of $g \colon B \mapsto \overline{C}$. Therefore, to formally interpret $f;g$, we silently restrict the domain of f to $A \times \Sigma_B$.

Lemma 5. $\ggg$ *is associative, left-distributive (but not right-distributive), has neutral element* δ *and preserves (sub-)probability and s-finite kernels.*

Product of Kernels, Product of Kernels in the Presence of Exception States. For s-finite kernels $\kappa \colon A \mapsto B$, $\kappa' \colon A \mapsto C$, we define the *product of kernels*, denoted by $\kappa \times \kappa' \colon A \mapsto B \times C$, as $(\kappa \times \kappa')(a)(S) = (\kappa(a) \times \kappa'(a))(S)$. For s-finite kernels $\kappa \colon A \mapsto \overline{B}$ and $\kappa' \colon A \mapsto \overline{C}$, we define the *lifted product of kernels*, denoted by $\kappa \overline{\times} \kappa' \colon A \mapsto \overline{B \times C}$, as $(\kappa \overline{\times} \kappa')(a)(S) = (\kappa(a) \overline{\times} \kappa'(a))(S)$. $\times$ and $\overline{\times}$ allow us to combine kernels to a joint kernel. Essentially, this definition reduces the product of kernels to the product of measures.

Lemma 6. $\times$ *and* $\overline{\times}$ *for kernels preserve (sub-)probability and s-finite kernels, are associative, left- and right-distributive.*

Binding Conventions. To avoid too many parentheses, we make use of some binding conventions, ordering (in decreasing binding strength) $\overline{\times}, \times, ;, \ggg, +$.

Summary. The most important concepts introduced in this section are exception states, records, Lebesgue integration, Fubini's theorem and (s-finite) kernels.

4 A Probabilistic Language and Its Semantics

We now describe our probabilistic programming language, the typing rules and the denotational semantics of our language.

4.1 Syntax

Let $\mathbb{V} := \mathbb{Q} \cup \{\pi, e\} \subseteq \mathbb{R}$ be a (countable) set of constants expressible in our programs. Let $i, n \in \mathbb{N}$, $r \in \mathbb{V}$, $x \in \text{Vars}$, $\ominus$ a generic unary operator (e.g., $-$ inverts the sign of a value, ! is logical negation mapping 0 to 1 and all other numbers to 0, $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ round down and up respectively), $\oplus$ a generic binary operator (e.g., $+, -, *, /,$ ^ for addition, subtraction, multiplication, division and exponentiation, &&, || for logical conjunction and disjunction, $=, \neq, <, \leq, >, \geq$ to compare values). Let $f \colon A \to \mathbb{R} \to [0, \infty)$ be a measurable function that maps $a \in A$ to a probability density function. We check if f is measurable by uncurrying f to $f \colon A \times \mathbb{R} \to [0, \infty)$. Fig. 2 shows the syntax of our language.

Our expressions capture () (the only element of $\mathbb{1}$), r (real numbers), x (variables), $(e_1, \dots, e_n)$ (tuples), $e[i]$ (accessing elements of tuples for $i \in \mathbb{N}$), $\ominus e$ (unary operators), $e_1 \oplus e_2$ (binary operators), $e_1[e_2]$ (accessing array elements), $e_1[e_2 \mapsto e_3]$ (updating array elements), **array**(e_1, e_2) (creating array of length e_1

$$\begin{aligned} e &::= () \mid r \mid x \mid (e_1, \ldots, e_n) \mid e[i] \mid \ominus e \mid e_1 \oplus e_2 \mid e_1[e_2] \mid && \text{(Expressions)} \\ &\quad \textbf{array}(e_1, e_2) \mid e_1[e_2 \mapsto e_3] \mid F(e) \\ F &::= \lambda x.\{P; \textbf{return } e;\} \mid \textbf{flip} \mid \textbf{uniform} \mid \textbf{sampleFrom}_f && \text{(Functions)} \\ P &::= \textbf{skip} \mid x := e \mid x = e \mid P_1; P_2 \mid \textbf{if } e \, \{P_1\} \textbf{ else } \{P_2\} \mid \{P\} \mid && \text{(Statements)} \\ &\quad \textbf{assert}(e) \mid \textbf{observe}(e) \mid \textbf{while } e \, \{P\} \end{aligned}$$

Fig. 2. The syntax of our probabilistic language.

containing e_2 at every index) and $F(e)$ (evaluating function F on argument e). To handle functions $F(e_1, \ldots, e_n)$ with multiple arguments, we interpret $(e_1, \ldots, e_n)$ as a tuple and apply F to that tuple.

Our functions express $\lambda x.\{P; \textbf{return } e;\}$ (function taking argument x running P on x and returning e), $\textbf{flip}(e)$ (random choice from $\{0, 1\}$, 1 with probability e), $\textbf{uniform}(e_1, e_2)$ (continuous uniform distribution between e_1 and e_2) and $\textbf{sampleFrom}_f(e)$ (sample value distributed according to probability density function $f(e)$). An example for f is the density of the exponential distribution, indexed with rate λ. Formally, $f\colon (0, \infty) \to \mathbb{R} \to [0, \infty)$ is defined by $f(\lambda)(x) = \lambda e^{-\lambda x}$ if $x \geq 0$ and $f(\lambda)(x) = 0$ otherwise. Often, f is partial (e.g., $\lambda \leq 0$ is not allowed). Intuitively, arguments outside the allowed range of f produce the error state $\bot$.

Our statements express **skip** (no operation), $x := e$ (assigning to a fresh variable), $x = e$ (assigning to an existing variable), $P_1; P_2$ (sequential composition of programs), **if** e $\{P_1\}$ **else** $\{P_2\}$ (if-then-else), $\{P\}$ (static scoping), $\textbf{assert}(e)$ (asserting that an expression evaluates to true, assertion failure results in $\bot$), $\textbf{observe}(e)$ (observing that an expression evaluates to true, observation failure results in $\lightning$) and **while** e $\{P\}$ (while loops, non-termination results in $\circlearrowleft$). We additionally introduce syntactic sugar $e_1[e_2] = e_3$ for $e_1 = e_1[e_2 \mapsto e_3]$, **if** (e) $\{P\}$ for **if** e $\{P\}$ **else** $\{\textbf{skip}\}$ and $\texttt{func}(e_2)$ for $\lambda x.\{P; \textbf{return } e_1;\}(e_2)$ (using the name `func` for the function with argument x and body $\{P; \textbf{return } e_1\}$).

4.2 Typing Judgments

Let $n \in \mathbb{N}$. We define types by the following grammar in BNF, where $\tau[]$ denotes arrays over type τ. We sometimes write $\prod_{i=1}^{n} \tau_i$ for the product type $\tau_1 \times \cdots \times \tau_n$.

$$\tau ::= \mathbb{1} \mid \mathbb{R} \mid \tau[] \mid \tau_1 \times \cdots \times \tau_n$$

Note that we also use the type $\tau_1 \mapsto \tau_2$ of kernels with source τ_1 and target τ_2, but we do not list it here to avoid higher-order functions (discussed in Sect. 4.5).

Formally, a *context* Γ is a set $\{x_i \colon \tau_i\}_{i \in [n]}$ that assigns a type τ_i to each variable $x_i \in$ Vars. In slight abuse of notation, we sometimes write $x \in \Gamma$ if there is a type τ with $x\colon \tau \in \Gamma$. We also write $\Gamma, x\colon \tau$ for $\Gamma \cup \{x\colon \tau\}$ (where $x \notin \Gamma$) and Γ, Γ' for $\Gamma \cup \Gamma'$ (where Γ and Γ' have no common variables).

$$\frac{}{\Gamma \vdash ()\colon \mathbb{1}} \qquad \frac{}{\Gamma \vdash r\colon \mathbb{R}}\ r \in \mathbb{V} \qquad \frac{}{\Gamma \vdash x\colon \tau}\ x\colon \tau \in \Gamma \qquad \frac{\Gamma \vdash e_1\colon \tau_1 \quad \cdots \quad \Gamma \vdash e_n\colon \tau_n}{\Gamma \vdash (e_1,\dots,e_n)\colon \tau_1 \times \cdots \times \tau_n}$$

$$\frac{\Gamma \vdash e\colon \tau_0 \times \cdots \times \tau_{n-1}}{\Gamma \vdash e[i]\colon \tau_i}\ i \in \{0,\dots,n-1\} \qquad \frac{\Gamma \vdash e\colon \mathbb{R}}{\Gamma \vdash \ominus e\colon \mathbb{R}} \qquad \frac{\Gamma \vdash e_1\colon \mathbb{R} \quad \Gamma \vdash e_2\colon \mathbb{R}}{\Gamma \vdash e_1 \oplus e_2\colon \mathbb{R}}$$

$$\frac{\Gamma \vdash e_1\colon \tau[\,] \quad \Gamma \vdash e_2\colon \mathbb{R}}{\Gamma \vdash e_1[e_2]\colon \tau} \qquad \frac{\Gamma \vdash e_1\colon \mathbb{R} \quad \Gamma \vdash e_2\colon \tau}{\Gamma \vdash \texttt{array}(e_1,e_2)\colon \tau[\,]}$$

$$\frac{\Gamma \vdash e_1\colon \tau[\,] \quad \Gamma \vdash e_2\colon \mathbb{R} \quad \Gamma \vdash e_3\colon \tau}{\Gamma \vdash e_1[e_2 \mapsto e_3]\colon \tau[\,]} \qquad \frac{\Gamma \vdash e\colon \tau_1 \quad \vdash F\colon \tau_1 \mapsto \tau_2}{\Gamma \vdash F(e)\colon \tau_2}$$

$$\frac{x\colon \tau_1 \overset{P}{\rightsquigarrow} \Gamma \quad \Gamma \vdash e\colon \tau_2}{\vdash \lambda x.\{P; \texttt{return}\ e;\}\colon \tau_1 \mapsto \tau_2} \qquad \frac{}{\vdash \texttt{flip}\colon \mathbb{R} \mapsto \mathbb{R}} \qquad \frac{}{\vdash \texttt{uniform}\colon \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R}}$$

$$\frac{}{\vdash \texttt{sampleFrom}_f\colon \tau \mapsto \mathbb{R}}\ f\colon A \to \mathbb{R} \to [0,\infty), A \in \Sigma_\tau$$

Fig. 3. The typing rules for expressions and functions in our language

$$\frac{}{\Gamma \overset{\texttt{skip}}{\rightsquigarrow} \Gamma} \qquad \frac{\Gamma \vdash e\colon \tau}{\Gamma \overset{x:=e}{\rightsquigarrow} \Gamma, x\colon \tau}\ x \notin \Gamma \qquad \frac{\Gamma \vdash e\colon \tau}{\Gamma \overset{x=e}{\rightsquigarrow} \Gamma}\ x\colon \tau \in \Gamma \qquad \frac{\Gamma \overset{P}{\rightsquigarrow} \Gamma' \quad \Gamma' \overset{Q}{\rightsquigarrow} \Gamma''}{\Gamma \overset{P;Q}{\rightsquigarrow} \Gamma''}$$

$$\frac{\Gamma \overset{P}{\rightsquigarrow} \Gamma'}{\Gamma \overset{\{P\}}{\rightsquigarrow} \Gamma} \qquad \frac{\Gamma \vdash e\colon \mathbb{R} \quad \Gamma \overset{P_1}{\rightsquigarrow} \Gamma' \quad \Gamma \overset{P_2}{\rightsquigarrow} \Gamma'}{\Gamma \overset{\texttt{if}\ e\ \{P_1\}\ \texttt{else}\ \{P_2\}}{\rightsquigarrow} \Gamma'} \qquad \frac{\Gamma \vdash e\colon \mathbb{R}}{\Gamma \overset{\texttt{assert}(e)}{\rightsquigarrow} \Gamma} \qquad \frac{\Gamma \vdash e\colon \mathbb{R}}{\Gamma \overset{\texttt{observe}(e)}{\rightsquigarrow} \Gamma}$$

$$\frac{\Gamma \vdash e\colon \mathbb{R} \quad \Gamma \overset{P}{\rightsquigarrow} \Gamma}{\Gamma \overset{\texttt{while}\ e\ \{P\}}{\rightsquigarrow} \Gamma}$$

Fig. 4. The typing rules for statements

The rules in Figs. 3 and 4 allow deriving the type of expressions, functions and statements. To state that an expression e is of type τ under a context Γ, we write $\Gamma \vdash e\colon \tau$. Likewise, $\vdash F\colon \tau \mapsto \tau'$ indicates that F is a kernel from τ to τ'. Finally, $\Gamma \overset{P}{\rightsquigarrow} \Gamma'$ states that a context Γ is transformed to Γ' by a statement P. For **sampleFrom**$_f$, we intuitively want f to map values from τ to probability density functions. To allow f to be partial, i.e., to be undefined for some values from τ, we use $A \in \Sigma_\tau$ (and hence $A \subseteq [\![\tau]\!]$) as the domain of f (see Sect. 4.3).

4.3 Semantics

Semantic Domains. We assign to each type τ a set $[\![\tau]\!]$ together with an implicit σ-algebra Σ_τ on that set. Additionally, we assign a set $[\![\Gamma]\!]$ to each context $\Gamma = \{x_i\colon \tau_i\}_{i\in[n]}$. Concretely, we have $[\![\mathbb{1}]\!] = \mathbb{1} := \{()\}$ with $\Sigma_{\mathbb{1}} = \{\emptyset, ()\}$, $[\![\mathbb{R}]\!] = \mathbb{R}$ and $\Sigma_{\mathbb{R}} = \mathcal{B}$. The remaining semantic domains are outlined in Fig. 5.

$$\begin{aligned}
[\![\tau[\,]]\!] &= \bigcup_{i\in\mathbb{N}} [\![\tau]\!]^i && \Sigma_{\tau[]} \text{ is generated by } \bigcup_{i\in\mathbb{N}} \left\{ \prod_{j=1}^{i} S_j \,\middle|\, S_j \in \Sigma_\tau \right\} \\
[\![\tau_1 \times \cdots \times \tau_n]\!] &= \prod_{i=1}^{n} [\![\tau_i]\!] && \Sigma_{\tau_1\times\cdots\times\tau_n} \text{ is generated by } \left\{ \prod_{i=1}^{n} S_i \,\middle|\, S_i \in \Sigma_{\tau_i} \right\} \\
[\![\Gamma]\!] &= \prod_{i=1}^{n} (x_i \colon [\![\tau_i]\!]) && \Sigma_{\Gamma} \text{ is generated by } \left\{ \prod_{i=1}^{n} (x_i \colon S_i) \,\middle|\, S_i \in \Sigma_{\tau_i} \right\}
\end{aligned}$$

Fig. 5. Semantic domains for types

$$[\![()]\!]_{\mathbb{1}}(\sigma)(S) = [() \in S] \qquad [\![r]\!]_{\mathbb{R}}(\sigma)(S) = [r \in S] \qquad [\![x]\!]_{\tau}(\sigma)(S) = [\sigma[x] \in S]$$

$$[\![(e_1, \ldots, e_n)]\!]_{\tau_1\times\cdots\times\tau_n} = [\![e_1]\!]_{\tau_1} \overline{\times} \cdots \overline{\times} [\![e_n]\!]_{\tau_n} \qquad [\![e[i]]\!]_{\tau_i} = [\![e]\!]_{\tau_1\times\cdots\times\tau_n} \ggg \lambda t.\delta(t[i])$$

$$\begin{aligned}
[\![e_1/e_2]\!]_{\mathbb{R}} &= [\![e_1]\!]_{\mathbb{R}} \overline{\times} [\![e_2]\!]_{\mathbb{R}} \ggg \lambda(x,y). \begin{cases} \delta(x/y) & y \neq 0 \\ \delta(\bot) & y = 0 \end{cases} \\
[\![e_1[e_2]]\!]_{\tau} &= [\![e_1]\!]_{\tau[]} \overline{\times} [\![e_2]\!]_{\mathbb{R}} \ggg \lambda(t,i). \begin{cases} \delta(t[i]) & i \in \mathbb{N}, i < |t| \\ \delta(\bot) & \text{otherwise} \end{cases} \\
[\![e_1[e_2 \mapsto e_3]]\!]_{\tau[]} &= [\![e_1]\!]_{\tau[]} \overline{\times} [\![e_2]\!]_{\mathbb{R}} \overline{\times} [\![e_3]\!]_{\tau} \ggg \lambda(t,i,v). \begin{cases} \delta(t[i \mapsto v]) & i \in \mathbb{N}, i < |t| \\ \delta(\bot) & \text{otherwise} \end{cases} \\
[\![\texttt{array}(e_1, e_2)]\!]_{\tau[]} &= [\![e_1]\!]_{\mathbb{R}} \overline{\times} [\![e_2]\!]_{\tau} \ggg \lambda(n,v). \begin{cases} \delta(v!n) & n \in \mathbb{N} \\ \delta(\bot) & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 6. The semantics of expressions. $v!n$ stands for the n-tuple $(v, \ldots, v)$. $t[i]$ stands for the i-th element (0-indexed) of the tuple t and $t[i \mapsto v]$ is the tuple t, where the i-th element is replaced by v. $|t|$ is the length of a tuple t. σ stands for a program state over all variables in some Γ, with $\sigma \in [\![\Gamma]\!]$.

Expressions. Fig. 6 assigns to each expression e typed by $\Gamma \vdash e \colon \tau$ a probability kernel $[\![e]\!]_\tau \colon [\![\Gamma]\!] \mapsto \overline{[\![\tau]\!]}$. When τ is irrelevant or clear from the context, we may drop it and write $[\![e]\!]$. The formal interpretation of $[\![\Gamma]\!] \mapsto \overline{[\![\tau]\!]}$ is explained in Sect. 3.[3] Note that Fig. 6 is incomplete, but extending it is straightforward. When we need to evaluate multiple terms (as in $(e_1, \ldots, e_n)$), we combine the results using $\overline{\times}$. This makes sure that in the presence of exceptions, the first exception that occurs will have priority over later exceptions. In addition, deterministic functions (like $x+y$) are lifted to probabilistic functions by the Dirac delta (e.g. $\delta(x+y)$) and incomplete functions (like x/y) are lifted to complete functions via the explicit error state $\bot$.

[3] As a quick and intuitive reminder, $\kappa \colon A \mapsto \overline{B}$ means that for every $a \in A$, $\kappa(a)$ will be a distribution over $\overline{B}$, where $\overline{B}$ is B enriched with exception states. Hence, $\kappa(a)$ may have weight on elements of B, on exception states, or on both.

Fig. 7 assigns to each function F typed by $\vdash F\colon \tau_1 \mapsto \tau_2$ a probability kernel $[\![F]\!]_{\tau_1\mapsto\tau_2}\colon [\![\tau_1]\!] \mapsto \overline{[\![\tau_2]\!]}$. In the semantics of **`flip`**, $\delta(1)\colon \Sigma_{\overline{\mathbb{R}}} \to [0,\infty]$ is a measure on $\overline{\mathbb{R}}$, and $p \cdot \delta(1)$ rescales this measure pointwise. Similarly, the sum $p\cdot\delta(1) + (1-p)\cdot\delta(0)$ is also meant pointwise, resulting in a measure on $\overline{\mathbb{R}}$. Finally, $\lambda p.\, p\cdot\delta(1)+(1-p)\cdot\delta(0)$ is a kernel with source $[0,1]$ and target $\overline{\mathbb{R}}$. For $\texttt{sampleFrom}_f(e)$, remember that $f(p)(\cdot)$ is a probability density function.

$$
\begin{aligned}
[\![\texttt{flip}]\!]_{\mathbb{R}\mapsto\mathbb{R}} &= \lambda p.\begin{cases} p\cdot\delta(1)+(1-p)\cdot\delta(0) & p\in[0,1]\\ \delta(\bot) & \text{otherwise}\end{cases}\\
[\![\texttt{uniform}]\!]_{\mathbb{R}\mapsto\mathbb{R}} &= \lambda(l,r).\begin{cases} \lambda S.\frac{1}{r-l}\lambda([l,r]\cap S) & l<r\\ \delta(\bot) & \text{otherwise}\end{cases}\\
[\![\texttt{sampleFrom}_f]\!]_{\tau\mapsto\mathbb{R}} &= \lambda p.\begin{cases} \lambda S.\int_{x\in\mathbb{R}\cap S} f(p)(x)\lambda(dx) & p\in A\\ \delta(\bot) & p\notin A\end{cases}\\
[\![\lambda x.\{P;\texttt{return } e;\}]\!]_{\tau_1\mapsto\tau_2} &= \lambda v.\delta\left(\{x\mapsto v\}\right) >\!\!=\!\!> [\![P]\!] >\!\!=\!\!> [\![e_2]\!]_{\tau_2}
\end{aligned}
$$

Fig. 7. The semantics of functions.

$$
[\![\texttt{skip}]\!] = \delta \qquad [\![x := e]\!] = [\![x = e]\!] = \delta\overline{\times}[\![e]\!] >\!\!=\!\!> \lambda(\sigma,v).\delta(\sigma[x\mapsto v])
$$

$$
[\![P_1;P_2]\!] = [\![P_1]\!] >\!\!=\!\!> [\![P_2]\!] \qquad [\![\{P\}]\!] = [\![P]\!] >\!\!=\!\!> \lambda\sigma'.\delta(\sigma'(\Gamma))
$$

$$
[\![\texttt{if } e\ \{P_1\}\ \texttt{else}\ \{P_2\}]\!] = \delta\overline{\times}[\![e]\!]_{\mathbb{R}} >\!\!=\!\!> \lambda(\sigma,b).\begin{cases}[\![P_1]\!](\sigma) & b\neq 0\\ [\![P_2]\!](\sigma) & b=0\end{cases}
$$

$$
[\![\texttt{assert}(e)]\!] = \delta\overline{\times}[\![e]\!]_{\mathbb{R}} >\!\!=\!\!> \lambda(\sigma,b).\begin{cases}\delta(\sigma) & b\neq 0\\ \delta(\bot) & b=0\end{cases}
$$

$$
[\![\texttt{observe}(e)]\!] = \delta\overline{\times}[\![e]\!]_{\mathbb{R}} >\!\!=\!\!> \lambda(\sigma,b).\begin{cases}\delta(\sigma) & b\neq 0\\ \delta(\lightning) & b=0\end{cases}
$$

Fig. 8. The semantics of programs in our probabilistic language. Here, $\sigma[x\mapsto v]$ results in σ with the value stored under x updated to v. $\sigma'(\Gamma)$ selects only those variables from σ' that occur in Γ, meaning $\{x_i\mapsto v_i\}_{i\in\mathcal{I}}(\{x_i\colon\tau_i\}_{i\in\mathcal{I}'}) = \{x_i\mapsto v_i\}_{i\in\mathcal{I}\cap\mathcal{I}'}$.

Statements. Fig. 8 assigns to each statement P with $\Gamma \overset{P}{\rightsquigarrow} \Gamma'$ a probability kernel $[\![P]\!]\colon [\![\Gamma]\!] \mapsto \overline{[\![\Gamma']\!]}$. Note the use of $\overline{\times}$ in $\delta\overline{\times}[\![e]\!]$, which allows evaluating e while keeping the state σ in which e is being evaluated. Intuitively, if evaluating e results in an exception from $\mathcal{X}$, the previous state σ is irrelevant, and the result of $\delta\overline{\times}[\![e]\!]$ will be that exception from $\mathcal{X}$.

While Loop. To define the semantics of the while loop **`while`** e $\{P\}$, we introduce a *kernel transformer* $[\![$**`while`** e $\{P\}]\!]^{\mathsf{trans}}\colon ([\![\Gamma]\!] \mapsto \overline{[\![\Gamma]\!]}) \to ([\![\Gamma]\!] \mapsto \overline{[\![\Gamma]\!]})$ that transforms the semantics for n runs of the loop to the semantics for $n+1$ runs of the loop. Concretely,

$$[\![\texttt{while}\ e\ \{P\}]\!]^{\mathsf{trans}}(\kappa) = \delta\overline{\times}[\![e]\!] \ggg= \lambda(\sigma,b).\begin{cases}[\![P]\!](\sigma) \gg= \kappa & b \neq 0\\ \delta(\sigma) & b = 0\end{cases}$$

This semantics first evaluates e, while keeping the program state around using δ. If e evaluates to 0, the while loop terminates and we return the current program state σ. If e does not evaluate to 0, we run the loop body P and feed the result to the next iteration of the loop, using κ.

We can then define the semantics of **`while`** e $\{P\}$ using a special fixed point operator $\mathsf{fix}\colon ((A \mapsto \overline{A}) \to (A \mapsto \overline{A})) \to (A \mapsto \overline{A})$, defined by the pointwise limit $\mathsf{fix}(\Delta) = \lim_{n\to\infty} \Delta^n(\circlearrowleft)$, where $\circlearrowleft := \lambda\sigma.\,\delta(\circlearrowleft)$ and Δ^n denotes the n-fold composition of Δ. $\Delta^n(\circlearrowleft)$ puts all runs of the while loop that do not terminate within n steps into the state $\circlearrowleft$. In the limit, $\circlearrowleft$ only has weight on those runs of the loop that never terminate. $\mathsf{fix}(\Delta)$ is only defined if its pointwise limit exists. Making use of fix, we can define the semantics of the while loop as follows:

$$[\![\texttt{while}\ e\ \{P\}]\!] = \mathsf{fix}\Big([\![\texttt{while}\ e\ \{P\}]\!]^{\mathsf{trans}}\Big)$$

Lemma 7. *For Δ as in the semantics of the while loop, and for each σ and each S, the limit $\lim_{n\to\infty} \Delta^n(\circlearrowleft)(\sigma)(S)$ exists.*

Lemma 7 holds because increasing n may only shift probability mass from $\circlearrowleft$ to other states (we provide a formal proof in Appendix B). Kozen shows a different way of defining the semantics of the while loop [23], using least fixed points. Lemma 8 describes the relation of the semantics of our while loop to the semantics of the while loop of [23]. For more details on the formal interpretation of Lemma 8 and for its proof, see Appendix B.

Lemma 8. *In the absence of exception states, and using sub-probability kernels instead of distribution transformers, the definition of the semantics of the while loop from [23] is equivalent to ours.*

Theorem 2. *The semantics of each expression $[\![e]\!]$ and statement $[\![P]\!]$ is indeed a probability kernel.*

Proof. The proof proceeds by induction. Some lemmas that are crucial for the proof are listed in Appendix C. Conveniently, most functions that come up in our definition are continuous (like $a+b$) or continuous except on some countable subset (like $\frac{a}{b}$) and thus measurable.

4.4 Recursion

To extend our language with recursion, we apply the same ideas as for the while loop. Given the source code of a function F that uses recursion, we define its

$$\delta\overline{\times}\left[\!\left[\texttt{!flip}\left(\frac{1}{2}\right) \right]\!\right] \ggg \lambda(\sigma,b).\begin{cases}\Big(\kappa\overline{\times}[\![1]\!] \ggg \lambda(x,y).\,\delta\big(x+y\big)\Big)(\sigma) & b\neq 0\\ [\![0]\!](\sigma) & b=0\end{cases}$$

Fig. 9. Kernel transformer $[\![\texttt{geom}]\!]^{\mathsf{trans}}(\kappa)$ for `geom` given in Listing 11.

semantics in terms of a kernel transformer $[\![F]\!]^{\mathsf{trans}}$. This kernel transformer takes semantics for F up to a recursion depth of n and returns semantics for F up to recursion depth $n+1$. Formally, $[\![F]\!]^{\mathsf{trans}}(\kappa)$ follows the usual semantics, but uses κ as the semantics for recursive calls to F (we will provide an example shortly). Finally, we define the semantics of F by $[\![F]\!] := \mathsf{fix}\Big([\![F]\!]^{\mathsf{trans}}\Big)$. Just as for the while loop, $\mathsf{fix}\Big([\![F]\!]^{\mathsf{trans}}\Big)$ is well-defined because stepping from recursion depth n to $n+1$ can only shift probability mass from $\circlearrowleft$ to other states. We note that we could generalize our approach to mutual recursion.

To demonstrate how we define the kernel transformer, consider the recursive implementation of the geometric distribution in Listing 11 (to simplify presentation, Listing 11 uses early return). Given semantics κ for `geom` : $\mathbb{1} \mapsto \mathbb{R}$ up to recursion depth n, we can define the semantics of `geom` up to recursion depth $n+1$, as illustrated in Fig. 9.

```
geom(){
    if !flip(1/2){
        return geom()+1;
    }else{
        return 0;
    }
}
```

Listing 11. Geometric distribution

4.5 Higher-Order Functions

Our language cannot express higher-order functions. When trying to give semantics to higher-order probabilistic programs, an important step is to define a σ-algebra on the set of functions from real numbers to real numbers. Unfortunately, no matter which σ-algebra is picked, function evaluation (i.e. the function that takes f and x as arguments and returns $f(x)$) is not measurable [1]. This is a known limitation that previous work has looked into (e.g. [35] address it by restricting the set of functions to those expressible by their source code).

A promising recent approach is replacing measurable spaces by quasi-Borel spaces [16]. This allows expressing higher-order functions, at the price of replacing the well-known and well-understood measurable spaces by a new concept.

4.6 Non-determinism

To extend our language with non-determinism, we may define the semantics of expressions, functions and statements in terms of sets of kernels. For an expression e typed by $\Gamma \vdash e : \tau$, this means that $[\![e]\!]_\tau \in \mathcal{P}\left([\![\Gamma]\!] \mapsto [\![\tau]\!]\right)$, where $\mathcal{P}\left(S\right)$ denotes the power set of S. Lifting our semantics to non-determinism is mostly straightforward, except for loops. There, $[\![\texttt{while}\ e\ \{P\}]\!]$ contains all kernels of the form $\lim_{n\to\infty}(\Delta_1 \circ \cdots \circ \Delta_n)(\circlearrowleft)$, where $\Delta_i \in [\![\texttt{while}\ e\ \{P\}]\!]^{\mathsf{trans}}$. Previous work has studied non-determinism in more detail, see e.g. [21,22].

5 Properties of Semantics

We now investigate two properties of our semantics: commutativity and associativity. These are useful in practice, e.g. because they enable rewriting programs to a form that allows for more efficient inference [5].

In this section, we write $e_1 \simeq e_2$ when expressions e_1 and e_2 are equivalent (i.e. when $[\![e_1]\!] = [\![e_2]\!]$). Analogously, we write $P_1 \simeq P_2$ for $[\![P_1]\!] = [\![P_2]\!]$.

5.1 Commutativity

In the presence of exception states, our language cannot guarantee commutativity of expressions such as $e_1 + e_2$. This is not surprising, as in our semantics the first exception bypasses all later exceptions.

Lemma 9. *For function* $F()\{\texttt{while}\ 1\ \{\texttt{skip}\}; \texttt{return}\ 0\}$*,*

$$\frac{1}{0} + F() \not\simeq F() + \frac{1}{0}$$

Formally, this is because if we evaluate $\frac{1}{0}$ first, we only have weight on $\perp$. If instead, we evaluate $F()$ first, we only have weight on $\circlearrowleft$, by an analogous calculation. A more detailed proof is included in Appendix D.

However, the only reason for non-commutativity is the presence of exceptions. Assuming that e_1 and e_2 cannot produce exceptions, we obtain commutativity:

Lemma 10. *If* $[\![e_1]\!](\sigma)(\mathcal{X}) = [\![e_2]\!](\sigma)(\mathcal{X}) = 0$ *for all* σ*, then* $e_1 \oplus e_2 \simeq e_2 \oplus e_1$*, for any commutative operator* $\oplus$*.*

The proof of Lemma 10 (provided in Appendix D) relies on the absence of exceptions and Fubini's Theorem. This commutativity result is in line with the results from [34], which proves commutativity in the absence of exceptions.

In the analogous situation for statements, we cannot assume commutativity $P_1; P_2 \simeq P_2; P_1$, even if there is no dataflow from P_1 to P_2. We already illustrated this in Listing 10, where swapping two lines changes the program semantics. However, in the absence of exceptions and dataflow from P_1 to P_2, we can guarantee $P_1; P_2 \simeq P_2; P_1$.

5.2 Associativity

A careful reader might suspect that since commutativity does not always hold in the presence of exceptions, a similar situation might arise for associativity of some expressions. As an example, can we guarantee $e_1 + (e_2 + e_3) \simeq (e_1 + e_2) + e_3$, even in the presence of exceptions? The answer is yes, intuitively because exceptions can only change the behavior of a program if the order of their occurrence is changed. This is not the case for associativity. Formally, we derive the following:

Lemma 11. $e_1 \oplus (e_2 \oplus e_3) \simeq (e_1 \oplus e_2) \oplus e_3$*, for any associative operator* $\oplus$*.*

We include notes on the proof of Lemma 11 in Appendix D, mainly relying on the associativity of $\overline{\times}$ (Lemma 6). Likewise, sequential composition is associative: $(P_1; P_2); P_3 \simeq P_1; (P_2; P_3)$. This is due to the associativity of $>\!\!=\!\!>$ (Lemma 5).

5.3 Adding the **score** Primitive

Some languages include the primitive **score**, which allows to increase or decrease the probability of a certain event (or trace) [34,35].

Listing 12 shows an example program using **score**. Without normalization, it returns 0 with probability $\frac{1}{2}$ and 1 with "probability" $\frac{1}{2} \cdot 2 = 1$. After normalization, it returns 0 with probability $\frac{1}{3}$ and 1 with probability $\frac{2}{3}$. Because **score** allows decreasing the probability of a specific event, it renders **observe** unnecessary. In general, we can replace **observe**(e) by **score**$(e \neq 0)$. However, performing this replacement means losing the explicit knowledge of the weight on $\lightning$.

```
x:=flip(½);
if x=1 {
    score(2);
}
return x;
```

Listing 12. Using **score**

score can be useful to modify the shape of a given distribution. For example, Listing 13 turns the distribution of x, which is a Gaussian distribution, into the Lebesgue measure λ, by multiplying the density of x by its inverse. Hence, the density of x at any location is 1. Note that the distribution over x cannot be described by a probability measure, because e.g. the "probability" that x lies in the interval $[0, 2]$ is 2.

```
x:=gauss(0,1);
score($\sqrt{2\pi}e^{x^2/2}$);
return x;
```

Listing 13. Reshaping a distribution.

Unfortunately, termination in the presence of **score** is not well-defined, as illustrated in Listing 14. In this program, the only non-terminating trace keeps changing its weight, switching between 1 and 2. In the limit, it is impossible to determine the weight of non-termination.

Hence, allowing the use of the **score** primitive only makes sense after abolishing the tracking of non-termination ($\circlearrowleft$), which can be achieved by only measuring sets that do not contain non-termination. Formally, this means restricting the semantics of expressions e typed by $\Gamma \vdash e : \tau$ to $[\![e]\!]_\tau : \Gamma \mapsto \left(\overline{[\![\tau]\!]} - \{\circlearrowleft\}\right)$. Intuitively, abolishing non-termination means that we ignore non-terminating runs (these result in weight on non-termination). After doing this, we can give well-defined semantics to the **score** primitive.

```
i:=0;
while 1 {
    if i=0 {
        score(2);
    }else{
        score(½);
    }
    i=1-i;
}
```

Listing 14. **score** vs non-termination

The typing rule and semantics of **score** are:

$$\frac{\Gamma \vdash e : \mathbb{R}}{\Gamma \overset{\textbf{score}(e)}{\rightsquigarrow} \Gamma} \qquad \text{and} \qquad [\![\textbf{score}(e)]\!] = \delta \overline{\times} [\![e]\!]_{\mathbb{R}} \ggg \lambda(\sigma, c).c * \delta(\sigma)$$

After including **score** into our language, the semantics of the language can no longer be expressed in terms of probability kernels as stated in Theorem 2, because the probability of any event can be inflated beyond 1. Instead, the semantics must be expressed in terms of s-finite kernels.

Theorem 3. *After adding the* **score** *primitive and abolishing non-termination, the semantics of each expression* $[\![e]\!]$ *and statement* $[\![P]\!]$ *is an s-finite kernel.*

Table 3. Comparison of existing semantics to ours. When adding **score** to our language (Sect. 5.3), our semantics use s-finite kernels (not probability kernels).

Work	Language	Semantics	Typed	Higher-order	Loops	Constraints
We	Imperative	Probability kernels	Typed	First-order	Loops (FP)	Yes
[4]	Functional	Sub-probability kernels	Untyped	Higher-order	Recursion (FP)	Yes
[23]	Imperative	Distribution transformers	Limited	First-order	Loops (LFP)	No
[24]	Imperative	Sub-probability kernels	Limited	First-order	Loops (LFP)	Yes
[28]	Imperative	Weakest precondition	Untyped	First-order	Loops (LFP)	Yes
[33]	Declarative	Probability kernels	Limited	First-order	Loops (LFP)	No
[34]	Functional	s-finite kernels	Typed	First-order	Counting measure	**score**(x)
[35]	Functional	Measurable functions	Typed	Higher-order	No	**score**(x)

Proof. As for Theorem 2, the proof proceeds by induction. Most parts of the proof are analogous (e.g. $>\!\!=\!\!>$ preserves s-finite kernels instead of probability kernels). For while loops, the limit still exists (Lemma 7 still holds), but it is not bounded from above anymore. The limit indeed corresponds to an s-finite kernel because the limit of strictly increasing s-finite kernels is an s-finite kernel.

In the presence of **score**, we can still talk about the interaction of different exceptions, assuming that we do track different types of exceptions (e.g. division by zero and out of bounds access of arrays). Then, we keep the commutativity and associativity properties studied in the previous sections, because these still hold for s-finite kernels.

```
score(2);
assert(false);
```

Listing 15. Interaction of **score** and **assert**

Listing 15 shows an interaction of **score** with **assert**. As one would expect, our semantics will assign weight 2 to $\perp$ in this case. If the two statements are switched, our semantics will ignore **score**(2) and assign weight 1 to $\perp$. Hence again, commutativity does not hold.

```
while 1 {
    score(2);
    assert(flip(1/2));
}
```

Listing 16. Interaction of **score**, **assert** and loops

Listing 16 shows a program that keeps increasing the probability of an error. In every loop iteration, there is a "probability" of 1 of running into an error. Overall, Listing 16 results in weight ∞ on state $\perp$.

6 Related Work

Kozen provides classic semantics to probabilistic programs [23]. We follow his main ideas, but deviate in some aspects in order to introduce additional features or to make our presentation cleaner. The semantics by Hur et al. [19] is heavily based on [23], so we do not go into more detail here. Table 3 summarizes the comparison of our approach to that of others.

Kernels. Like our work, most modern approaches use kernels (i.e., functions from values to distributions) to provide semantics to probabilistic programs [4,24,33,34]. Borgström et al. [4] use sub-probability kernels on (symbolic) expressions.

Staton [34] uses s-finite kernels to capture the semantics of the **score** primitive (when we discuss **score** in Sect. 5.3, we do the same).

In the classic semantics of [23], Kozen uses distribution transformers (i.e., functions from distributions to distributions). For later work [24], Kozen also switches to sub-probability kernels, which has the advantage of avoiding redundancies. A different approach uses weakest precondition to define the semantics, as in [28]. Staton et al. [35] use a different concept of measurable functions $A \to P(\mathbb{R}_{\geq 0} \times B)$ (where $P(S)$ denotes the set of all probability measures on S).

Typing. Some probabilistic languages are untyped [4,28], while others are limited to just a single type: $\mathbb{R}^n$ [23,24] or $\bigcup_{i=1}^{\infty} \mathbb{N}^i \cup \mathbb{N}^\infty$ [33]. Some languages provide more interesting types including sum types, distribution types and tuples [34,35]. We allow tuples and array types, and we could easily account for sum types.

Loops. Because the semantics of while loops is not always straightforward, some languages avoid while loops and recursion altogether [35]. Borgström et al. handle recursion instead of while loops, defining the semantics in terms of a fixed point [4]. Many languages handle while loops by least fixed points [23,24,28,33]. Staton defines while loops in terms of the counting measure [34], which is similar to defining them by a fixed point. We define the semantics of while loops in terms of a fixed point, which avoids the need to prove the least fixed point exists (still, the classic while loop semantics of [23] and our formulation are equivalent).

Most languages do not explicitly track non-termination, but lose probability weight by non-termination [4,23,24,34]. This missing weight can be used to identify the probability of non-termination, but only if other exceptions (such as **fail** in [24] or observation failure in [4]) do not also result in missing weight. The semantics of [33] are tailored to applications in networks and lose *non-terminating packet histories* instead of weight (due to a particular least fixed point construction of Scott-continuous maps on algebraic and continuous directed complete partial orders). Some works define non-termination as missing weight in the weakest precondition [28]. Specifically, the semantics in [28] can also explicitly express probability of non-termination *or* ending up in some state (using the separate construct of a weakest liberal precondition). We model non-termination by an explicit state $\circlearrowleft$, which has the advantage that in the context of lost weight, we know what part of that lost weight is due to non-termination.

Kaminski et al. [21] investigate the run-time of probabilistic program with loops and **fail** (interpreted as early termination), but without observations. In [21], non-termination corresponds to an infinite run-time.

Error States. Many languages do not consider partial functions (like fractions $\frac{a}{b}$) and thus never run into an exception state [23,24,33]. Olmedo et al. [28] do not consider partial functions, but support the related concept of an explicit **abort**. The semantics of **abort** relies on missing weight in the final distribution. Some languages handle expressions whose evaluation may fail using sum types [34,35], forcing the programmer to deal with errors explicitly (we discuss the disadvantages of this approach at Listing 6). Formally, a sum type $A + B$ is a disjoint union of the two sets A and B. Defining the semantics of an expression in terms of the sum type $A+\{\bot\}$ allows that expression to evaluate to *either* a value $a \in A$ *or* to $\bot$. Borgström et al. [4] have a single state **fail** expressing exceptions such as dynamically detected type errors (without forcing the programmer to deal with exceptions explicitly). Our semantics also uses sum types to handle exceptions, but the handling is implicit, by defining semantics in terms of $(>=>)$ (which defines how exceptions propagate in a program) instead of $(;)$.

Constraints. To enforce hard constraints, we use the **observe**(e) statement, which puts the program into a special failure state $\lightning$ if it does not satisfy e. We can encode soft constraints by **observe**(e), where e is probabilistic (this is a general technique). Borgström et al. [4] allow both soft constraints that reduce the probability of some program traces and hard constraints whose failure leads to the error state **fail**. Some languages can handle generalized soft constraints: they can not only decrease the probability of certain traces using soft constraints, but also increase them, using **score**(x) [34,35]. We investigate the consequences of adding **score** to our language in Sect. 5.3. Kozen [24] handles hard (and hence soft) constraints using **fail** (which results in a sub-probability distribution). Some languages can handle neither hard nor soft constraints [23,33]. Note though that the semantics of ProbNetKAT in [33] can drop certain packages, which is a similar behavior. Olmedo et al. [28] handle hard (and hence soft) constraints by a conditional weakest precondition that tracks both the probability of not failing any observation and the probability of ending in specific states. Unfortunately, this work is restricted to discrete distributions and is specifically designed to handle observation failures and non-termination. Thus, it is not obvious how to adapt the semantics if a different kind of exception is to be added.

Interaction of Different Exceptions. Most existing work handles at least some exceptions by sub-probability distributions [4,23,24,33,34]. Then, any missing weight in the final distribution must be due to exceptions. However, this leads to a conflation of all exceptions handled by sub-probability distributions (for the consequences of this, see, e.g., our discussion of Listing 8). Note that semantics based on sub-probability kernels can add more exceptions, but they will simply be conflated with all other exceptions.

Some previous work does not (exclusively) rely on sub-probability distributions. Borgström et al. [4] handle errors implicitly, but still use sub-probability kernels to handle non-termination and **score**. Olmedo et al. can distinguish non-termination (which is conflated with exception failure) from failing observations by introducing two separate semantic primitives (conditional weakest precondition and conditional liberal weakest precondition) [28]. Because their solution specifically addresses non-termination, it is non-trivial to generalize this treatment to more than two exception states. By using sum types, some semantics avoid interactions of errors with non-termination or constraint failures, but still cannot distinguish the latter [34,35]. Note that semantics based on sum types can easily add more exceptions (although it is impossible to add non-termination).

However, the interaction of different exceptions cannot be observed, because the programmer has to handle exceptions explicitly.

To the best of our knowledge, we are the first to give formal semantics to programs that may produce exceptions in this generality. One work investigates assertions in probabilistic programs, but explicitly disallows non-terminating loops [32]. Moreover, the semantics in [32] are operational, leaving the distribution (in terms of measure theory) of program outputs unclear. Cho et al. [8] investigate the interaction of partial programs and observe, but are restricted to discrete distributions and to only two exception states. In addition, this investigation treats these two exception states differently, making it non-trivial to extend the results to three or more exception states. Katoen et al. [22] investigate the intuitive problems when combining non-termination and observations, but restrict their discussions to discrete distributions and do not provide formal semantics. Huang [17] treats partial functions, but not different kinds of exceptions. In general, we know of no probabilistic programming language that distinguishes more than two different kinds of exceptions. Distinguishing two kinds of exceptions is simpler than three, because it is possible to handle one exception as an explicit exception state and the other one by missing weight (as e.g. in [4]).

Cousot and Monerau [9] provide a trace semantics that captures probabilistic behavior by an explicit randomness source given to the program as an argument. This allows handling non-termination by non-terminating traces. While the work does not discuss errors or observation failure, it is possible to add both. However, using an explicit randomness source has other disadvantages, already discussed by Kozen [23]. Most notably, this approach requires a distribution over the randomness source and a translation from the randomness source to random choices in the program, even though we only care about the distribution of the latter.

7 Conclusion

In this work we presented an expressive probabilistic programming language that supports important features such as mixing continuous and discrete distributions, arrays, observations, partial functions and while-loops. Unlike prior work, our semantics distinguishes non-termination, observation failures and error states. This allows us to investigate the subtle interaction of different exceptions, which is not possible for semantics that conflate different kinds of exceptions. Our investigation confirms the intuitive understanding of the interaction of exceptions presented in Sect. 2. However, it also shows that some desirable properties, like commutativity, only hold in the absence of exceptions. This situation is unavoidable, and largely analogous to the situation in deterministic languages.

Even though our semantics only distinguish three exception states, it can be trivially extended to handle any countable set of exception states. This allows for an even finer-grained distinction of e.g. division by zero, out of bounds array accesses or casting failures (in a language that allows type casting). Our semantics also allows enriching exceptions with the line number that the exception originated from (of course, this is not possible for non-termination). For an uncountable set of exception states, an extension is possible but not trivial.

A Proofs for Preliminaries

In this section, we provide lemmas, proofs and some definitions that were left out or cut short in Sect. 3. For a more detailed introduction into measure theory, we recommend the book *A crash course on the Lebesgue integral and measure theory* [7].

A.1 Measures

Definition 3. *Let (A, Σ_A) be a measurable space and $\mu\colon \Sigma_A \to [0, \infty]$ a measure on A.*

- *We call μ s-finite if μ can be written as a countable sum $\sum_{i \in \mathbb{N}} \mu_i$ of sub-probability measures μ_i.*
- *We call μ σ-finite if $A = \bigcup_{i \in \mathbb{N}} A_i$ for $A_i \in \Sigma_A$, with $\mu(A_i) < \infty$.*
- *We call μ finite if $\mu(A) < \infty$.*
- *We call μ a sub-probability measure if $\mu(A) \leq 1$.*
- *We call μ a probability measure if $\mu(A) = 1$.*

Note that for a σ-finite measure μ, $\mu(A) = \infty$ is possible, even though $\mu(A_i) < \infty$ for all i. As an example, the Lebesgue measure is σ-finite because $\mathbb{R} = \bigcup_{i \in \mathbb{N}} [-i, i]$ with $\lambda([-i, i]) = 2 * i$, but $\lambda(\mathbb{R}) = \infty$.

Lemma 12. *The following definition of s-finite measures is equivalent to our definition of s-finite measures (the difference is that the μ_is are only required to be finite):*

We call $\mu\colon \Sigma_A \to [0, \infty]$ an s-finite measure if it can be written as $\mu = \sum_{i \in \mathbb{N}} \mu_i$ for finite measures $\mu_i\colon \Sigma_A \to [0, \infty]$.

Proof. Since any sub-probability measure is finite, one direction is trivial. For the other direction, let $\mu = \sum_{i \in \mathbb{N}} \mu_i'$ for finite measures μ_i'. Obviously, $\mu \geq 0$, $\mu(\emptyset) = 0$ and $\mu(\bigcup_{i \in \mathbb{N}} A_i) = \sum_{i \in \mathbb{N}} A_i$ for mutually disjoint $A_i \in \Sigma_A$, so μ is a measure. To show that μ can be written as a sum of sub-probability measures, let $n_i := \lceil \mu_i'(A) \rceil$. Then, $\mu = \sum_{i \in \mathbb{N}} \mu_i' = \sum_{i \in \mathbb{N}} \frac{n_i}{n_i} \mu_i' = \sum_{i \in \mathbb{N}} \sum_{j \in [n_i]} \frac{1}{n_i} \mu_i'$. We let $\mu_i := \frac{1}{n_i} \mu_i' \leq 1$.

Lemma 13. *Any σ-finite measure $\mu\colon \Sigma_A \to [0, \infty]$ is s-finite.*

Proof. Since μ is σ-finite, $A = \bigcup_{i \in \mathbb{N}} A_i$ with $A_i \in \Sigma_A$ and $\mu(A_i) < \infty$. Without loss of generality, assume that the A_i form a partition of A. Then, $\mu(S) = \sum_{i \in \mathbb{N}} \mu(S \cap A_i)$, with $\mu(\cdot \cap A_i) < \infty$. Thus, μ is a countable sum of finite measures.

Definition 4. *The counting measure* $c\colon \mathcal{B} \to [0,\infty]$ *is defined by*

$$c(S) = \begin{cases} |S| & S \text{ finite} \\ \infty & \text{otherwise} \end{cases}$$

Definition 5. *The infinity measure* $\mu\colon \mathcal{B} \to [0,\infty]$ *is defined by*

$$\mu(S) = \begin{cases} 0 & S = \emptyset \\ \infty & \text{otherwise} \end{cases}$$

Lemma 14. *Neither the counting measure nor the infinity measure are s-finite.*

Proof. For the counting measure c, assume (toward a contradiction) $c = \sum_{i\in\mathbb{N}} c_i$. We have $\mathbb{R} = \{r \in \mathbb{R} \mid c(\{r\}) > 0\} = \bigcup_{i\in\mathbb{N}}\{r \in \mathbb{R} \mid c_i(\{r\}) > 0\} = \bigcup_{i\in\mathbb{N}}\bigcup_{n\in\mathbb{N}}\{r \in \mathbb{R} \mid c_i(\{r\}) > \frac{1}{n}\}$. Because $\mathbb{R}$ is uncountable, there must be $i, n \in \mathbb{N}$ for which $S := \{r \in \mathbb{R} \mid c_i(\{r\}) > \frac{1}{n}\}$ is uncountable. Thus for any measurable, countably infinite $S' \subseteq S$, $c_i(S') = \infty$, which means that c_i is not finite. Proceed analogously for the infinity measure.

Lemma 15. *The measure* $\mu : \mathcal{B} \to [0,\infty]$ *with* $\mu(S) = \left\{\begin{matrix} 0 & \lambda(S) = 0 \\ \infty & \lambda(S) > 0 \end{matrix}\right\}$ *is s-finite but not* σ*-finite.*

Proof. $\mu = \sum_{i\in\mathbb{N}} \lambda$, and λ is s-finite, so μ is s-finite. Assume (toward a contradiction) that μ is σ-finite. Then $\mathbb{R} = \bigcup_{i\in\mathbb{N}} A_i$ with $A_i \in \mathcal{B}$ and $\mu(A_i) < \infty$. Thus, $\mu(A_i) = 0$ and hence $\mu(\mathbb{R}) = \mu(\bigcup_{i\in\mathbb{N}} A_i) \le \sum_{i\in\mathbb{N}} \mu(A_i) = 0$, a contradiction.

Lemma 16

$$\begin{aligned}\forall S \in \Sigma_{A\times B}\colon (\mu \times \mu')(S) &= \int_{a\in A} \mu'(\{b \in B \mid (a,b) \in S\})\mu(da) \\ &= \int_{b\in B} \mu(\{a \in A \mid (a,b) \in S\})\mu'(db)\end{aligned}$$

$$\begin{aligned}\forall S \in \Sigma_{\overline{A\times B}}\colon (\mu \overline{\times} \mu')(S) &= \int_{a\in \overline{A}} \mu'(\{b \in \overline{B} \mid \overline{(a,b)} \in S\})\mu(da) \\ &= \int_{b\in \overline{B}} \mu(\{a \in \overline{A} \mid \overline{(a,b)} \in S\})\mu'(db)\end{aligned}$$

Proof

$$\begin{aligned}(\mu \times \mu')(S) &= \int\limits_{a\in A}\int\limits_{b\in B} [(a,b) \in S]\mu'(db)\mu(da) \\ &= \int\limits_{a\in A}\int\limits_{b\in B} [b \in \{b' \in B \mid (a,b') \in S\}]\mu'(db)\mu(da) \\ &= \int\limits_{a\in A} \mu'(\{b' \in B \mid (a,b') \in S\})\mu(da)\end{aligned}$$

$$\begin{aligned}(\mu\times\mu')(S) &= \int\limits_{a\in A}\int\limits_{b\in B}[(a,b)\in S]\mu'(db)\mu(da) \\ &= \int\limits_{b\in B}\int\limits_{a\in A}[(a,b)\in S]\mu(da)\mu'(db) && \text{Fubini} \\ &= \dots \\ &= \int\limits_{b\in B}\mu(\{a'\in A\mid (a',b)\in S\})\mu'(db)\end{aligned}$$

In the second line, we have used that $(a,b)\in S \iff b\in\{b'\in B\mid (a,b')\in S\}$. The proof works analogously for $\overline{\times}$.

Lemma 17. *Let $\delta\colon A\mapsto A$, $\kappa\colon A\mapsto B$. Then,*

$$(\delta\overline{\times}\kappa)(a)(S)=\kappa(a)(\{b\in\overline{B}\mid\overline{(a,b)}\in S\})$$

Proof

$$\begin{aligned}(\delta\overline{\times}\kappa)(a)(S) &= \int_{b\in\overline{B}}\delta(a)(\{a'\in A\mid\overline{(a',b)}\in S\})\kappa(a)(db) && \text{Lemma 16} \\ &= \int_{b\in\overline{B}}[\overline{(a,b)}\in S]\kappa(a)(db) \\ &= \kappa(a)(\{b\in\overline{B}\mid\overline{(a,b)}\in S\})\end{aligned}$$

Lemma 1. *For measures $\mu\colon\Sigma_A\to[0,\infty]$, $\mu'\colon\Sigma_B\to[0,\infty]$, let $S\in\Sigma_A$ and $T\in\Sigma_B$. Then, $(\mu\times\mu')(S\times T)=\mu(S)\cdot\mu'(T)$.*

Proof

$$\begin{aligned}(\mu\times\mu')(S\times T) &= \int_{a\in A}\mu'(\{b\in B\mid (a,b)\in S\times T\})\mu(da) && \text{Lemma 16} \\ &= \int_{a\in A}\mu'\left(\left\{\begin{matrix}T & a\in S\\ \emptyset & \text{otherwise}\end{matrix}\right\}\right)\mu(da) \\ &= \int_{a\in S}\mu'(T)\mu(da) \\ &= \mu(S)*\mu'(T)\end{aligned}$$

Lemma 2. *$\times$ and $\overline{\times}$ for s-finite measures are associative, left- and right-distributive and preserve (sub-)probability and s-finite measures.*

Proof. Remember that $(\mu\times\mu')(S)=\int_{a\in A}\int_{b\in B}[(a,b)\in S]\mu'(db)\mu(da)$ and that $(\mu\overline{\times}\mu')(S)=\int_{a\in\overline{A}}\int_{b\in\overline{B}}[\overline{(a,b)}\in S]\mu'(db)\mu(da)$. Preservation of (sub-)probability measures is trivial. Distributivity and preservation of s-finite measures are easily established by properties of the Lebesgue integral in Lemma 19.

For associativity, let $\mu\colon\Sigma_A\to[0,\infty]$, $\mu\colon\Sigma_B\to[0,\infty]$ and $\mu\colon\Sigma_C\to[0,\infty]$.

$$\begin{aligned}
&((\mu \times \mu') \times \mu'')(S) \\
=&\int_{c\in C} (\mu \times \mu')(\{t \in A \times B \mid (t,c) \in S\})\mu''(dc) && \text{Lemma 16} \\
=&\int_{c\in C}\int_{a\in A}\int_{b\in B} [(a,b) \in \{t \in A \times B \mid (t,c) \in S\}]\mu'(db)\mu(da)\mu''(dc) \\
=&\int_{c\in C}\int_{a\in A}\int_{b\in B} [(a,b,c) \in S]\mu'(db)\mu(da)\mu''(dc) \\
=&\int_{a\in A}\int_{b\in B}\int_{c\in C} [(a,b,c) \in S]\mu''(dc)\mu'(db)\mu(da) && \text{Fubini} \\
=&\int_{a\in A}\int_{b\in B}\int_{c\in C} [(b,c) \in \{t \in B \times C \mid (a,t) \in S\}]\mu''(dc)\mu'(db)\mu(da) \\
=&\int_{a\in A} (\mu' \times \mu'')(\{t \in B \times C \mid (a,t) \in S\})\mu(da) \\
=&(\mu \times (\mu' \times \mu''))(S)\mu(da) && \text{Lemma 16}
\end{aligned}$$

The proof proceeds analogously for $\overline{\times}$.

Lemma 18. *Let* (A, Σ_A) *and* (B, Σ_B) *be measurable spaces. Consider measures* $\mu, \mu_1, \mu_2 \colon \Sigma_A \to [0, \infty]$ *and* $\nu, \nu_1, \nu_2 \colon \Sigma_B \to [0, \infty]$*. We assume that* $\nu_1 \leq \nu_2$ *and* $\mu_1 \leq \mu_2$ *hold pointwise. Then,*

$$\mu \overline{\times} \nu_1 \leq \mu \overline{\times} \nu_2$$
$$\mu_1 \overline{\times} \nu \leq \mu_2 \overline{\times} \nu$$

Proof. Let $S \in \Sigma_{A\times B}$ and $\nu_1 \leq \nu_2$. Then, we have

$$\begin{aligned}
&\nu_1 \leq \nu_2 \\
\implies &\underbrace{\int_{b\in\overline{B}} [\overline{(a,b)} \in S]\nu_1(db)}_{=:f(a)} \leq \underbrace{\int_{b\in\overline{B}} [\overline{(a,b)} \in S]\nu_2(db)}_{=:g(a)} && \text{Lemma 19} \\
\implies &\int_{a\in\overline{A}} f(a)\mu(da) \leq \int_{a\in\overline{A}} g(a)\mu(da) && \text{Lemma 19} \\
\implies &(\mu \overline{\times} \nu_1)(S) \leq (\mu \overline{\times} \nu_2)(S)
\end{aligned}$$

The proof for $\mu_1 \overline{\times} \nu \leq \mu_2 \overline{\times} \nu$ is similar.

A.2 Lebesgue Integral

Lemma 19. *Let* (A, Σ_A) *and* (B, Σ_B) *be measurable spaces,* $E \in \Sigma_A$ *and* $E' \in \Sigma_B$ *measurable sets,* $f, f_i, g \colon A \to \mathbb{R}$ *and* $h \colon A \times B \to \mathbb{R}$ *measurable functions,*

$\mu, \mu_i, \nu \colon \Sigma_A \to [0,\infty]$ and $\mu' \colon \Sigma_B \to [0,\infty]$ measures.

$$\int\limits_{a\in E} f(a)\mu(da) \in [0,\infty]$$

$$0 \le f \le g \le \infty \implies \int\limits_{a\in E} f(a)\mu(da) \le \int\limits_{a\in E} g(a)\mu(da)$$

$$\mu \le \nu \implies \int\limits_{a\in E} f(a)\mu(da) \le \int\limits_{a\in E} f(a)\nu(da)$$

$$\sum_{n=1}^{\infty} \int\limits_{a\in E} f_n(a)\mu(da) = \int\limits_{a\in E} \sum_{n=1}^{\infty} f_n(a)\mu(da)$$

$$\int\limits_{a\in E} \int\limits_{b\in E'} f(a,b)\mu'(db)\mu(da) = \int\limits_{b\in E'} \int\limits_{a\in E} f(a,b)\mu'(da)\mu(db) \qquad \mu, \mu' \sigma\text{-finite}$$

$$\int\limits_{a\in E} f(a)\left(\sum_{n=1}^{\infty} \mu_i\right)(da) = \sum_{n=1}^{\infty} \int\limits_{a\in E} f(a)\mu_i(da)$$

$$\int\limits_{a\in E} f(a)\delta(x)(da) = f(x) \qquad x \in E$$

Finally, if $f_1 \le f_2 \le \cdots \le \infty$, we have

$$\lim_{n\to\infty} \int\limits_{a\in E} f_n(a)\mu(da) = \int\limits_{a\in E} \lim_{n\to\infty} f_n(a)\mu(da)$$

Proof. The following properties can be proven for simple functions and limits of simple functions (this suffices):

$$\int_{a\in E} f(a)\left(\sum_{n=1}^{\infty} \mu_i\right)(da) = \sum_{n=1}^{\infty} \int_{a\in E} f(a)\mu_i(da)$$

$$\mu \le \nu \implies \int_{a\in E} f(a)\mu(da) \le \int_{a\in E} f(a)\nu(da)$$

$\int_{a\in E} f(a)\delta(x)(da) = f(x)$ is straightforward. For the other properties, see [31].

Theorem 1 (Fubini's theorem). *For s-finite measures $\mu \colon \Sigma_A \to [0,\infty]$ and $\mu' \colon \Sigma_B \to [0,\infty]$ and any measurable function $f \colon A \times B \to [0,\infty]$,*

$$\int_{a\in A} \int_{b\in B} f(a,b)\mu'(db)\mu(da) = \int_{b\in B} \int_{a\in A} f(a,b)\mu(da)\mu'(db)$$

For s-finite measures $\mu \colon \Sigma_{\overline{A}} \to [0,\infty]$ and $\mu' \colon \Sigma_{\overline{B}} \to [0,\infty]$ and any measurable function $f \colon A \times B \to [0,\infty]$,

$$\int_{a\in\overline{A}} \int_{b\in\overline{B}} \overline{f}(a,b)\mu'(db)\mu(da) = \int_{b\in\overline{B}} \int_{a\in\overline{A}} \overline{f}(a,b)\mu(da)\mu'(db)$$

Proof. Let $\mu = \sum_{i\in\mathbb{N}} \mu_i$ and $\mu' = \sum_{i\in\mathbb{N}} \mu'_i$ for bounded measures μ_i and μ'_i.

$$\begin{aligned}
&\int_{a\in A}\int_{b\in B} f(a,b)\mu'(db)\mu(da) \\
&= \sum_{i,j\in\mathbb{N}} \int_{a\in A}\int_{b\in B} f(a,b)\mu'_j(db)\mu_i(da) \quad \text{Lemma 19} \\
&= \sum_{i,j\in\mathbb{N}} \int_{b\in B}\int_{a\in A} f(a,b)\mu_i(da)\mu'_j(db) \quad \text{Fubini for}\sigma\text{-finite measures } \mu_i, \mu'_j \\
&= \int_{b\in B}\int_{a\in A} f(a,b)\mu(da)\mu'(db)
\end{aligned}$$

The proof in the presence of exception state is analogous.

Lemma 20. *Fubini does not hold for the counting measure $c\colon \mathcal{B} \to [0,\infty]$ and the Lebesgue measure $\lambda\colon \mathcal{B} \to [0,\infty]$ (because c is not s-finite).*

Proof

$$\int_{x\in[0,1]}\int_{y\in[0,1]} [x=y]c(dy)\lambda(dx) = \int_{x\in[0,1]} 1\lambda(dx) = 1$$
$$\int_{y\in[0,1]}\int_{x\in[0,1]} [x=y]\lambda(dx)c(dy) = \int_{y\in[0,1]} 0c(dy) = 0$$

A.3 Kernels

Lemma 21. *Let $\kappa_1, \kappa'_1 \colon A \mapsto \overline{B}$ and $\kappa_2, \kappa'_2 \colon B \mapsto \overline{C}$ be s-finite kernels.*
If $\kappa_1 \le \kappa'_1$ holds pointwise, then

$$\kappa_1 \ggg \kappa_2 \le \kappa'_1 \ggg \kappa_2$$

If $\kappa_2 \le \kappa'_2$ holds pointwise, then

$$\kappa_1 \ggg \kappa_2 \le \kappa_1 \ggg \kappa'_2$$

Proof. Assume $\kappa_2 \le \kappa'_2$. Thus, $\overline{\kappa_2} \le \overline{\kappa'_2}$. Now, let $a \in A$, $S \in \Sigma_{\overline{C}}$.

$$\begin{aligned}
(\kappa_1 \ggg \kappa_2)(a)(S) &= \int_{b\in\overline{B}} \overline{\kappa_2}(b)(S)\,\kappa_1(a)(db) \\
&\le \int_{b\in\overline{B}} \overline{\kappa'_2}(b)(S)\,\kappa_1(a)(db) \qquad \overline{\kappa_2} \le \overline{\kappa'_2}, \text{Lemma 19} \\
&= (\kappa_1 \ggg \kappa'_2)(a)(S)
\end{aligned}$$

The proof for $\kappa_1 \ggg \kappa_2 \le \kappa'_1 \ggg \kappa_2$ works analogously.

Lemma 3. $(;)$ *is associative, left- and right-distributive, has neutral element*[4] δ *and preserves (sub-)probability and s-finite kernels.*

[4] δ is a neutral element of $(;)$ if $(\delta;\kappa) = (\kappa;\delta) = \kappa$ for all kernels κ.

Proof. Remember that $(f;g)(a)(S) = \int_{b \in B} g(b)(S)\, f(a)(db)$. Left- and right-distributivity and the neutral element δ follow from properties of the Lebesgue integral in Lemma 19.

Associativity and preservation of (sub-)probability kernels is well known (see for example [12]). For s-finite kernels $f = \sum_{i \in \mathbb{N}} f_i$ and $g = \sum_{i \in \mathbb{N}} g_i$ and $h = \sum_{i \in \mathbb{N}} h_i$, we have (for sub-probability kernels f_i, g_i, h_i)

$$(f;g);h = \left(\left(\sum_{i \in \mathbb{N}} f_i\right);\left(\sum_{j \in \mathbb{N}} g_j\right)\right);\sum_{k \in \mathbb{N}} h_k = \sum_{i,j,k \in \mathbb{N}} (f_i;g_j);h_k$$
$$= \sum_{i,j,k \in \mathbb{N}} f_i;(g_j;h_k) = f;(g;h)$$

(;) preserves s-finite kernels because for s-finite kernels f and g, we have (for sub-probability kernels f_i, g_i) $f;g = \sum_{i,j \in \mathbb{N}} f_i;g_i$, a sum of kernels.

Lemma 4. *For $f\colon A \mapsto \overline{B}$ and $g\colon B \mapsto \overline{C}$, $a \in A$ and $S \in \Sigma_{\overline{C}}$,*

$$(f \ggg g)(a)(S) = (f;g)(a)(S) + \sum_{x \in \mathcal{X}} \delta(x)(S) f(a)(\{x\})$$

Proof

$$\begin{aligned}(f \ggg g)(a)(S) &= \int_{b \in \overline{B}} \overline{g}(b)(S)\, f(a)(db) \\ &= \int_{b \in B} \overline{g}(b)(S)\, f(a)(db) + \int_{b \in \mathcal{X}} \overline{g}(b)(S)\, f(a)(db) \\ &= \int_{b \in B} g(b)(S)\, f(a)(db) + \sum_{b \in \mathcal{X}} \overline{g}(b)(S)\, f(a)(\{x\}) \\ &= (f;g)(a)(S) + \sum_{x \in \mathcal{X}} \delta(x)(S) f(a)(\{x\})\end{aligned}$$

Lemma 5. *$\ggg$ is associative, left-distributive (but not right-distributive), has neutral element δ and preserves (sub-)probability and s-finite kernels.*

Proof. Remember that $(f \ggg g)(a)(S) = \int_{b \in \overline{B}} \overline{g}(b)(S)\, f(a)(db)$. Left-distributivity follows from the properties of the Lebesgue integral in Lemma 19. Right-distributivity does not necessarily hold because $\overline{g_1 + g_2}(\perp) \neq \overline{g_1}(\perp) + \overline{g_2}(\perp)$. Associativity for $f\colon A \mapsto \overline{B}$, $g\colon B \mapsto \overline{C}$ and $h\colon C \mapsto \overline{D}$ can be derived by

$$
\begin{aligned}
&((f >\!\!=\!\!> g) >\!\!=\!\!> h)(a)(S)\\
&= \Big(\Big(f >\!\!=\!\!> g\Big);h\Big)(a)(S) + \sum_{x\in\mathcal{X}} \delta(x)(S)(f >\!\!=\!\!> g)(a)(\{x\})\\
&= \Big(\Big(f;g + \lambda a'.\lambda S'.\sum_{x\in\mathcal{X}} \delta(x)(S')f(a')(\{x\})\Big);h\Big)(a)(S)\\
&\quad + \sum_{x\in\mathcal{X}} \delta(x)(S)(f >\!\!=\!\!> g)(a)(\{x\})\\
&= (f;g;h)(a)(S) + \underbrace{\Big(\Big(\lambda a'.\lambda S'.\sum_{x\in\mathcal{X}} \delta(x)(S')f(a')(\{x\})\Big);h\Big)(a)(S)}_{=0 ((;)\text{integrates over non-exception states})}\\
&\quad + \sum_{x\in\mathcal{X}} \delta(x)(S)(f >\!\!=\!\!> g)(a)(\{x\})\\
&= (f;g;h)(a)(S) + \sum_{x\in\mathcal{X}} \delta(x)(S)\Big((f;g)(a)(\{x\}) + \sum_{x'\in\mathcal{X}} \delta(x')(\{x\})f(a)(\{x'\})\Big)\\
&= (f;g;h)(a)(S) + \sum_{x\in\mathcal{X}} \delta(x)(S)\Big((f;g)(a)(\{x\}) + f(a)(\{x\})\Big)\\
&= (f;g;h)(a)(S) + \sum_{x\in\mathcal{X}} \delta(x)(S)(f;\lambda a'.\lambda S'.g(a')(S'))(a)(\{x\})\\
&\quad + \sum_{x\in\mathcal{X}} \delta(x)(S)f(a)(\{x\})\\
&= (f;g;h)(a)(S) + \Big(f;\Big(\lambda a'.\lambda S'.\sum_{x\in\mathcal{X}} \delta(x)(S')g(a')(\{x\})\Big)\Big)(a)(S)\\
&\quad + \sum_{x\in\mathcal{X}} \delta(x)(S)f(a)(\{x\})\\
&= \Big(f;\Big(g;h + \lambda a'.\lambda S'.\sum_{x\in\mathcal{X}} \delta(x)(S')g(a')(\{x\})\Big)\Big)(a)(S) + \sum_{x\in\mathcal{X}} \delta(x)(S)f(a)(\{x\})\\
&= \Big(f;\Big(g >\!\!=\!\!> h\Big)\Big)(a)(S) + \sum_{x\in\mathcal{X}} \delta(x)(S)f(a)(\{x\})\\
&= (f >\!\!=\!\!> (g >\!\!=\!\!> h))(a)(S)
\end{aligned}
$$

Here, we have used Lemma 4, left- and right-distributivity of $(;)$.

To show that $f >\!\!=\!\!> g$ preserves s-finite kernels, let $f\colon A \mapsto \overline{B}$ and $g\colon B \mapsto \overline{C}$ be s-finite kernels. Then, for sub-probability kernels f_i,

$$
\begin{aligned}
(f >\!\!=\!\!> g)(a)(S) &= (f;g)(a)(S) + \sum_{x\in\mathcal{X}} \delta(x)(S)f(a)(\{x\})\\
&= (f;g)(a)(S) + \sum_{x\in\mathcal{X}} \sum_{i\in\mathbb{N}} \delta(x)(S)f_i(a)(\{x\})
\end{aligned}
$$

Note that for each $x \in \mathcal{X}$ and $i \in \mathbb{N}$, $\lambda a.\lambda S.\delta(x)(S)f_i(a)(\{x\})$ is a sub-probability kernel. Thus, $f >\!\!=\!\!> g$ is a sum of s-finite kernels and hence s-finite.

Proving that for sub-probability kernels f and g, $f \mathbin{>\!\!=\!\!>} g$ is also a (sub-)probability kernel is trivial, since we only need to show that $(f \mathbin{>\!\!=\!\!>} g)(a)(\overline{C}) = 1$ (or ≤ 1).

Lemma 22. *Let (A, Σ_A) and (B, Σ_B) be measurable spaces. Let $f\colon A \times B \to [0, \infty]$ be measurable and $\kappa\colon A \mapsto B$ be a sub-probability kernel. Then, $f'\colon A \to [0, \infty]$ defined by*

$$f'(a) := \int_{b \in B} f(a, b)\kappa(a)(db)$$

is measurable.

Proof. See Theorem 20 of [30].

Lemma 23. $\times$ *and* $\overline{\times}$ *preserve (sub-)probability kernels.*

Proof. Let $\kappa\colon A \mapsto B$ and $\kappa'\colon A \mapsto C$ be (sub-)probability kernels. The fact that $(\kappa \times \kappa')(a)(\cdot)$ for all $a \in A$ is a (sub-)probability measure is inherited from Lemma 2. It remains to show that $(\kappa \times \kappa')(\cdot)(S)$ is measurable for all $S \in \Sigma_{B \times C}$, with

$$(\kappa \times \kappa')(a)(S) = \int_{b \in B} \int_{c \in C} [(b, c) \in S]\kappa'(a)(dc)\kappa(a)(db)$$

By Lemma 22, $f'\colon A \times B \to [0, \infty]$ defined by $f'(a, b) = \int_{c \in C}[(b, c) \in S]\kappa'(a)(dc)$ is measurable, using the measurable function $f\colon (A \times B) \times C \to [0, \infty]$ defined by $f((a, b), c) = [(b, c) \in S]$. Again by Lemma 22, $\int_{b \in B} \int_{c \in C}[(b, c) \in S]\kappa'(a)(dc)\kappa(a)(db)$ is measurable.

Proving that for (sub-)probability kernels $\kappa\colon A \mapsto \overline{B}$ and $\kappa'\colon A \mapsto \overline{C}$, $\kappa \overline{\times} \kappa'$ is a (sub-)probability kernel proceeds analogously.

Lemma 6. $\times$ *and* $\overline{\times}$ *for kernels preserve (sub-)probability and s-finite kernels, are associative, left- and right-distributive.*

Proof. Associativity, left- and right-distributivity are inherited from respective properties of the product of measures established by Lemma 2. Sub-probability kernels are preserved by Lemma 23.

S-finite kernels are preserved because $\kappa \times \kappa' = (\sum_{i \in \mathbb{N}} \kappa_i) \times (\sum_{i \in \mathbb{N}} \kappa'_i) = \sum_{i,j \in \mathbb{N}} \kappa_i \times \kappa'_j$ (analogously for $\overline{\times}$).

B Proofs for Semantics

Lemma 7. *For Δ as in the semantics of the while loop, and for each σ and each S, the limit $\lim_{n\to\infty} \Delta^n(\circlearrowleft)(\sigma)(S)$ exists.*

Proof. In general, $0 \leq \Delta^n(\circlearrowleft)(\sigma)(S) \leq 1$. First, we restrict the allowed arguments for $\lim_{n\to\infty} \Delta^n(\circlearrowleft)(\sigma)(S)$ to only those S with $\circlearrowleft \in S$. We prove by induction that $\Delta^{n+1}(\circlearrowleft) \leq \Delta^n(\circlearrowleft)$, meaning $\forall\sigma\colon \forall S\colon \circlearrowleft \in S \implies \Delta^{n+1}(\circlearrowleft)(\sigma)(S) \leq \Delta^n(\circlearrowleft)(\sigma)(S)$. Hence, $\Delta^n(\circlearrowleft)$ is monotone decreasing in n and lower bounded by 0, which means that the limit must exist.

As a base case, we have $\Delta^1(\circlearrowleft)(\sigma)(S) \leq 1 = \delta_{\circlearrowleft}(S) = \Delta^0(\circlearrowleft)(\sigma)(S)$, because $\circlearrowleft \in S$. We proceed by induction with

$$\begin{aligned}\Delta^{n+1}(\circlearrowleft)(\sigma)(S) &= \left(\delta \overline{\times} [\![e]\!] >\!\!=\!\!> \lambda(\sigma, b). \begin{Bmatrix} [\![P]\!](\sigma) \gg\!= \Delta^n(\circlearrowleft) & b \neq 0 \\ \delta(\sigma) & b = 0 \end{Bmatrix}\right)(\sigma)(S) \\ &\leq \left(\delta \overline{\times} [\![e]\!] >\!\!=\!\!> \lambda(\sigma, b). \begin{Bmatrix} [\![P]\!](\sigma) \gg\!= \Delta^{n-1}(\circlearrowleft) & b \neq 0 \\ \delta(\sigma) & b = 0 \end{Bmatrix}\right)(\sigma)(S) \\ &= \Delta^n(\circlearrowleft)(\sigma)(S)\end{aligned}$$

In the second line, we have used the induction hypothesis. This application is valid because $\kappa_2 \leq \kappa_2'$ implies $\kappa_1 >\!\!=\!\!> \kappa_2 \leq \kappa_1 >\!\!=\!\!> \kappa_2'$ (Lemma 21).

We proceed analogously when we restrict the allowed arguments for the kernel $\lim_{n\to\infty} \Delta^n(\circlearrowleft)(\sigma)(S)$ to only those S with $\circlearrowleft \notin S$, proving $\Delta^{n+1}(\circlearrowleft) \geq \Delta^n(\circlearrowleft)$ for that case.

Lemma 8. *In the absence of exception states, and using sub-probability kernels instead of distribution transformers, the definition of the semantics of the while loop from [23] is equivalent to ours.*

Definition 6. *In [23], Kozen shows a different way of defining the semantics of the while loop. In our notation, and in terms of probability kernels instead of distribution transformers, that definition becomes*

$$[\![\texttt{while}\ e\ \{P\}]\!] = \sup_{n\in\mathbb{N}} \sum_{k=0}^{n} \Big([\![\texttt{filter}(e)]\!] >\!\!=\!\!> [\![P]\!]\Big)^k >\!\!=\!\!> [\![\texttt{filter}(\neg e)]\!]$$

Here, exponentiation is in terms of Kleisli composition, i.e. $\kappa^0 = \delta$ *and* $\kappa^{n+1} = \kappa >\!\!=\!\!> \kappa^n$. *The sum and limit are meant pointwise. Furthermore, we define filter by the following expression (note that* $[\![\texttt{filter}(e)]\!]$ *and* $[\![\texttt{filter}(\neg e)]\!]$ *are only sub-probability kernels, not probability kernels).*

$$[\![\texttt{filter}(e)]\!] = \delta \overline{\times} [\![e]\!] >\!\!=\!\!> \lambda(\sigma, b). \begin{Bmatrix} \delta(\sigma) & b \neq 0 \\ \mathbf{0} & b = 0 \end{Bmatrix}$$

$$[\![\texttt{filter}(\neg e)]\!] = \delta \overline{\times} [\![e]\!] >\!\!=\!\!> \lambda(\sigma, b). \begin{Bmatrix} \delta(\sigma) & b = 0 \\ \mathbf{0} & b \neq 0 \end{Bmatrix}$$

To justify Lemma 8, we prove the more formal Lemma 24. Note that in the presence of exceptions (e.g. P is just **assert**(0)), Definition 6 does not make sense, because if

Lemma 24. *For all S with $S \cap \mathcal{X} = \emptyset$*

$$\left(\sum_{k=0}^{n} \Big([\![\texttt{filter}(e)]\!] >\!\!=\!\!> [\![P]\!]\Big)^k >\!\!=\!\!> [\![\texttt{filter}(\neg e)]\!]\right)(\sigma)(S) = \Delta^{n+1}(\circlearrowleft)(\sigma)(S)$$

Proof. For $n = 0$, we have

$$
\begin{aligned}
&\left(\sum_{k=0}^{0}\left([\![\mathtt{filter}(e)]\!] >\!\!=\!\!=\!\!> [\![P]\!]\right)^{k} >\!\!=\!\!=\!\!> [\![\mathtt{filter}(\neg e)]\!]\right)(\sigma)(S)\\
&=\left(\left([\![\mathtt{filter}(e)]\!] >\!\!=\!\!=\!\!> [\![P]\!]\right)^{0} >\!\!=\!\!=\!\!> [\![\mathtt{filter}(\neg e)]\!]\right)(\sigma)(S)\\
&=\left(\delta >\!\!=\!\!=\!\!> [\![\mathtt{filter}(\neg e)]\!]\right)(\sigma)(S)\\
&=[\![\mathtt{filter}(\neg e)]\!](\sigma)(S)\\
&=\left(\delta\overline{\times}[\![e]\!] >\!\!=\!\!=\!\!> \lambda(\sigma',b).\begin{Bmatrix}\delta(\sigma') & b=0\\ \mathbf{0} & b\neq 0\end{Bmatrix}\right)(\sigma)(S)\\
&=\left(\delta\overline{\times}[\![e]\!] >\!\!=\!\!=\!\!> \lambda(\sigma',b).\begin{Bmatrix}\delta(\sigma') & b=0\\ \circlearrowleft(\sigma') & b\neq 0\end{Bmatrix}\right)(\sigma)(S) && \circlearrowleft\notin S\\
&=\left(\delta\overline{\times}[\![e]\!] >\!\!=\!\!=\!\!> \lambda(\sigma',b).\begin{Bmatrix}\delta(\sigma') & b=0\\ ([\![P]\!] >\!\!=\!\!=\!\!>\circlearrowleft)(\sigma') & b\neq 0\end{Bmatrix}\right)(\sigma)(S) && S\cap\mathcal{X}=\emptyset\\
&=\Delta^{1}(\circlearrowleft)
\end{aligned}
$$

For $n \geq 0$, we have

$$
\begin{aligned}
&\left(\sum_{k=0}^{n+1}\left([\![\mathtt{filter}(e)]\!] >\!\!=\!\!=\!\!> [\![P]\!]\right)^{k} >\!\!=\!\!=\!\!> [\![\mathtt{filter}(\neg e)]\!]\right)(\sigma)(S)\\
&=\left(\left(\sum_{k=0}^{n}\left([\![\mathtt{filter}(e)]\!] >\!\!=\!\!=\!\!> [\![P]\!]\right)^{k+1} + ([\![\mathtt{filter}(e)]\!] >\!\!=\!\!=\!\!> P)^{0}\right) >\!\!=\!\!=\!\!> [\![\mathtt{filter}(\neg e)]\!]\right)(\sigma)(S)\\
&=\left(\left(\sum_{k=0}^{n}\left([\![\mathtt{filter}(e)]\!] >\!\!=\!\!=\!\!> [\![P]\!]\right)^{k+1} + \delta\right) >\!\!=\!\!=\!\!> [\![\mathtt{filter}(\neg e)]\!]\right)(\sigma)(S)\\
&=\left(\left(\sum_{k=0}^{n}\left([\![\mathtt{filter}(e)]\!] >\!\!=\!\!=\!\!> [\![P]\!]\right)^{k+1}\right) >\!\!=\!\!=\!\!> [\![\mathtt{filter}(\neg e)]\!]\right)(\sigma)(S) && \text{since } S\cap\mathcal{X}=\emptyset\\
&\quad+\left(\delta >\!\!=\!\!=\!\!> [\![\mathtt{filter}(\neg e)]\!]\right)(\sigma)(S)\\
&=\left(\left(\sum_{k=0}^{n}\left([\![\mathtt{filter}(e)]\!] >\!\!=\!\!=\!\!> [\![P]\!]\right)^{k+1}\right) >\!\!=\!\!=\!\!> [\![\mathtt{filter}(\neg e)]\!]\right)(\sigma)(S) + [\![\mathtt{filter}(\neg e)]\!](\sigma)(S)\\
&=\left(\left([\![\mathtt{filter}(e)]\!] >\!\!=\!\!=\!\!> [\![P]\!] >\!\!=\!\!=\!\!> \sum_{k=0}^{n}([\![\mathtt{filter}(e)]\!] >\!\!=\!\!=\!\!> [\![P]\!])^{k}\right) >\!\!=\!\!=\!\!> [\![\mathtt{filter}(\neg e)]\!]\right)(\sigma)(S)\\
&\quad+[\![\mathtt{filter}(\neg e)]\!](\sigma)(S)\\
&=\left([\![\mathtt{filter}(e)]\!] >\!\!=\!\!=\!\!> [\![P]\!] >\!\!=\!\!=\!\!> \left(\sum_{k=0}^{n}([\![\mathtt{filter}(e)]\!] >\!\!=\!\!=\!\!> [\![P]\!])^{k} >\!\!=\!\!=\!\!> [\![\mathtt{filter}(\neg e)]\!]\right)\right)(\sigma)(S)\\
&\quad+[\![\mathtt{filter}(\neg e)]\!](\sigma)(S)\\
&=\left([\![\mathtt{filter}(e)]\!] >\!\!=\!\!=\!\!> [\![P]\!] >\!\!=\!\!=\!\!> \Delta^{n+1}(\circlearrowleft)\right)(\sigma)(S) + [\![\mathtt{filter}(\neg e)]\!](\sigma)(S)\\
&=\left(\delta\overline{\times}[\![e]\!] >\!\!=\!\!=\!\!> \lambda(\sigma',b).\begin{Bmatrix}[\![P]\!](\sigma') \gg= \Delta^{n+1}(\circlearrowleft) & b\neq 0\\ \delta(\sigma') & b=0\end{Bmatrix}\right)(\sigma)(S)\\
&=\Delta^{n+2}(\circlearrowleft)(\sigma)(S)
\end{aligned}
$$

In particular, have have used that left-distributivity does hold in this case since $S \cap \mathcal{X} = \emptyset$.

C Probability Kernel

In the following, we list lemmas that are crucial to prove Theorem 2 (restated for convenience).

Theorem 2. *The semantics of each expression $[\![e]\!]$ and statement $[\![P]\!]$ is indeed a probability kernel.*

Lemma 25. *Any measurable function $f\colon A \to [0,\infty]$ can be viewed as an s-finite kernel $f\colon A \mapsto \mathbb{1}$, defined by $f(x)(\emptyset) = 0$ and $f(x)(\mathbb{1}) = f(x)$.*

Proof. We prove that f is an s-finite kernel. Let $A_\infty := \{x \in A \mid f(x) = \infty\}$. Since f is measurable, the set A_∞ must be measurable. $f(x)(S) = \sum_{i\in\mathbb{N}}[x \in A_\infty][() \in S] + \sum_{i\in\mathbb{N}} f(x)[i \leq f(x) < i+1][() \in S]$, which is a sum of finite kernels because the sets A_∞ and $\{x \mid i \leq f(x) < i+1\} = f^{-1}([i, i+1))$ are measurable. Note that any sum of finite kernels can be rewritten as a sum of sub-probability kernels.

Lemma 26. *Let $\kappa'\colon X \mapsto Y$ and $\kappa''\colon X \mapsto Y$ be kernels, and $f\colon X \to \mathbb{R}$ measurable. Then,*

$$\kappa(x)(S) = \begin{cases} \kappa'(x)(S) & \textit{if } f(x) = 0 \\ \kappa''(x)(S) & \textit{otherwise} \end{cases}$$

is a kernel.

Proof. Let $f_{=0}(x) := [f(x) = 0]$, $f_{\neq 0}(x) := [f(x) \neq 0]$. Then, $\kappa = f_{=0} \times \kappa' + f_{\neq 0} \times \kappa''$. Viewing $f_{=0}$ and $f_{\neq 0}$ as kernels $X \mapsto \mathbb{1}$ immediately gives the desired result.

Lemma 27. *Let (A, Σ_A) and (B, Σ_B) be measurable spaces. Let $\{A_i\}_{i\in\mathcal{I}}$ be a partition of A into measurable sets, for a countable set of indices $\mathcal{I}$. Consider a function $f\colon A \to B$. If $f_{|A_i}\colon A_i \to B$ is measurable for each $i \in \mathcal{I}$, then f is measurable.*

Lemma 28. *Let $f\colon A \to B$ be measurable. Then $\kappa\colon A \mapsto B$ with $\kappa(a) = \delta(f(a))$ is a kernel.*

The following lemma is important to show that the semantics of the while loop is a probability kernel.

Lemma 29. *Suppose $\{\kappa_n\}_{n\in\mathbb{N}}$ is a sequence of (sub-)probability kernels $A \mapsto B$. Then, if the limit $\kappa = \lim_{n\to\infty} \kappa_n$ exists, it is also a (sub-)probability kernel. Here, the limit is pointwise in the sense $\forall a \in A\colon \forall S \in \Sigma_B\colon \kappa(a, S) = \lim_{n\to\infty} \kappa_n(a)(S)$.*

Proof. For every $a \in A$, $\kappa(a, \cdot)$ is a measure, because the pointwise limit of finite measures is a measure. For every $S \in \Sigma_B$, $\kappa(\cdot, S)$ is measurable, because the pointwise limit of measurable functions $f_n\colon A \to \mathbb{R}$ (with $\mathcal{B}$ as the σ-algebra on $\mathbb{R}$) is measurable.

D Proofs for Consequences

In this section, we provide some proofs of consequences of our semantics, explained in Sect. 5.

Lemma 9. *For function* $F()\{\texttt{while}\ 1\ \{\texttt{skip}\};\texttt{return}\ 0\}$,

$$\frac{1}{0} + F() \not\simeq F() + \frac{1}{0}$$

Proof. If we evaluate $\frac{1}{0}$ first, we will only have weight on $\bot$.

$$\begin{aligned}
&\left[\!\left[\frac{1}{0} + F()\right]\!\right] \\
=&\left[\!\left[\frac{1}{0}\right]\!\right]\overline{\times}[\![F()]\!] \ggg \lambda(x,y).\delta(x+y) \\
=&\,\delta(\bot)\overline{\times}[\![F()]\!] \ggg \lambda(x,y).\delta(x+y) \\
=&\,\delta(\bot) \ggg \lambda(x,y).\delta(x+y) \\
=&\,\delta(\bot)
\end{aligned}$$

If instead, we first evaluate $F()$, we only have weight on $\circlearrowleft$, by an analogous calculation.

Lemma 10. *If* $[\![e_1]\!](\sigma)(\mathcal{X}) = [\![e_2]\!](\sigma)(\mathcal{X}) = 0$ *for all* σ, *then* $e_1 \oplus e_2 \simeq e_2 \oplus e_1$, *for any commutative operator* $\oplus$.

Proof

$$\begin{aligned}
[\![e_1 \oplus e_2]\!](\sigma)(S) &= [\![e_1]\!]\overline{\times}[\![e_2]\!] \ggg \lambda(x,y).\delta(x \oplus y) \\
&= \int_{z\in\overline{\mathbb{R}\times\mathbb{R}}} \overline{\lambda(x,y).\delta(x \oplus y)}(z)(S)([\![e_1]\!]\overline{\times}[\![e_2]\!])(\sigma)(dz) \\
&= \int_{(x,y)\in\mathbb{R}\times\mathbb{R}} \delta(x \oplus y)(S)([\![e_1]\!] \times [\![e_2]\!])(\sigma)(d(x,y)) \\
&= \int_{(y,x)\in\mathbb{R}\times\mathbb{R}} \delta(y \oplus x)(S)([\![e_2]\!] \times [\![e_1]\!])(\sigma)(d(y,x)) \\
&= [\![e_2 \oplus e_1]\!](\sigma)(S)
\end{aligned}$$

Here, we crucially rely on the absence of exceptions (for the third equality) and Fubini's Theorem (for the fourth equality).

Lemma 11. $e_1 \oplus (e_2 \oplus e_3) \simeq (e_1 \oplus e_2) \oplus e_3$, *for any associative operator* $\oplus$.

Proof. The important steps of the proof are the following.

$$\begin{aligned}
[\![e_1 \oplus (e_2 \oplus e_3)]\!] &= [\![e_1]\!] \overline{\times} [\![e_2 \oplus e_3]\!] \mathrel{>\!\!=\!\!>} \lambda(x,s).\delta(x \oplus s) \\
&= [\![e_1]\!] \overline{\times} \Big([\![e_2]\!] \overline{\times} [\![e_3]\!] \mathrel{>\!\!=\!\!>} \lambda(y,z).\delta(y \oplus z) \Big) \mathrel{>\!\!=\!\!>} \lambda(x,s).\delta(x \oplus s) \\
&= [\![e_1]\!] \overline{\times} \Big([\![e_2]\!] \overline{\times} [\![e_3]\!] \Big) \mathrel{>\!\!=\!\!>} \lambda(x,(y,z)).\delta(x \oplus y \oplus z) \\
&= \Big([\![e_1]\!] \overline{\times} [\![e_2]\!] \Big) \overline{\times} [\![e_3]\!] \mathrel{>\!\!=\!\!>} \lambda((x,y),z).\delta(x \oplus y \oplus z) \\
&= [\![(e_1 \oplus e_2) \oplus e_3]\!]
\end{aligned}$$

Here, we make crucial use of associativity for the lifted product of measures in Lemma 6.

References

1. Aumann, R.J.: Borel structures for function spaces. Ill. J. Math. **5**(4), 614–630 (1961)
2. Barthe, G., Grégoire, B., Hsu, J., Strub, P.-Y.: Coupling proofs are probabilistic product programs. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, pp. 161–174. ACM, New York (2017)
3. Barthe, G., Köpf, B., Olmedo, F., Zanella Béguelin, S.: Probabilistic relational reasoning for differential privacy. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, pp. 97–110. ACM, New York (2012)
4. Borgström, J., Dal Lago, U., Gordon, A.D., Szymczak, M.: A lambda-calculus foundation for universal probabilistic programming. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, pp. 33–46. ACM, New York (2016)
5. Chaganty, A., Nori, A., Rajamani, S.: Efficiently sampling probabilistic programs via program analysis. In: Artificial Intelligence and Statistics, pp. 153–160 (2013)
6. Chauveau, D., Diebolt, J.: An automated stopping rule for mcmc convergence assessment. Comput. Stat. **3**(14), 419–442 (1999)
7. Cheng, S.: A crash course on the lebesgue integral and measure theory (2008)
8. Cho, K., Jacobs, B.: Kleisli semantics for conditioning in probabilistic programming (2017)
9. Cousot, P., Monerau, M.: Probabilistic abstract interpretation. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 169–193. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_9
10. Gehr, T., Misailovic, S., Vechev, M.: PSI: exact symbolic inference for probabilistic programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 62–83. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_4
11. Gelman, A., Lee, D., Guo, J.: Stan a probabilistic programming language for bayesian inference and optimization. J. Educ. Behav. Stat. **40**, 530–543 (2015)
12. Giry, M.: A categorical approach to probability theory. In: Banaschewski, B. (ed.) Categorical Aspects of Topology and Analysis. LNM, vol. 915, pp. 68–85. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0092872

13. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. In: UAI, pp. 220–229 (2008)
14. Goodman, N.D., Stuhlmüller, A.: The design and implementation of probabilistic programming languages (2014). http://dippl.org. Accessed 15 May 2017
15. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Proceedings of the on Future of Software Engineering (2014)
16. Heunen, C., Kammar, O., Staton, S., Yang, H.: A convenient category for higher-order probability theory. CoRR, abs/1701.02547 (2017)
17. Huang, D.E.: On programming languages for probabilistic modeling (2017). https://danehuang.github.io/papers/dissertation.pdf. Accessed 28 June 2017
18. Hur, C.-K., Nori, A.V., Rajamani, S.K., Samuel, S.: Slicing probabilistic programs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 133–144. ACM, New York (2014)
19. Hur, C.-K., Nori, A.V., Rajamani, S.K., Samuel, S.: A provably correct sampler for probabilistic programs. In: LIPIcs-Leibniz International Proceedings in Informatics, vol. 45. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
20. Kaminski, B.L., Katoen, J.-P.: On the hardness of almost–sure termination. In: Italiano, G.F., Pighizzini, G., Sannella, D.T. (eds.) MFCS 2015. LNCS, vol. 9234, pp. 307–318. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48057-1_24
21. Kaminski, B.L., Katoen, J.-P., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run–times of probabilistic programs. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 364–389. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_15
22. Katoen, J.-P., Gretz, F., Jansen, N., Kaminski, B.L., Olmedo, F.: Understanding probabilistic programs. In: Meyer, R., Platzer, A., Wehrheim, H. (eds.) Correct System Design. LNCS, vol. 9360, pp. 15–32. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23506-6_4
23. Kozen, D.: Semantics of probabilistic programs. In: Proceedings of the 20th Annual Symposium on Foundations of Computer Science, SFCS 1979, pp. 101–114. IEEE Computer Society, Washington, DC (1979)
24. Kozen, D.: A probabilistic pdl. In: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, STOC 1983, pp. 291–297. ACM, New York (1983)
25. Mansinghka, V., Selsam, D., Perov, Y.: Venture: a higher-order probabilistic programming platform with programmable inference. ArXiv e-prints, March 2014
26. Minka, T., Winn, J., Guiver, J., Webster, S., Zaykov, Y., Yangel, B., Spengler, A., Bronskill, J.: Infer.NET 2.5. Microsoft Research Cambridge (2013). http://research.microsoft.com/infernet
27. Narayanan, P., Carette, J., Romano, W., Shan, C., Zinkov, R.: Probabilistic inference by program transformation in Hakaru (system description). In: Kiselyov, O., King, A. (eds.) FLOPS 2016. LNCS, vol. 9613, pp. 62–79. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29604-3_5
28. Olmedo, F., Gretz, F., Jansen, N., Kaminski, B.L., Katoen, J.-P., McIver, A.: Conditioning in probabilistic programming. ACM Trans. Program. Lang. Syst. (2018, to appear)
29. Paige, B., Wood, F.: A compilation target for probabilistic programming languages. In: International Conference on Machine Learning, pp. 1935–1943 (2014)
30. Pollard, D.: A User's Guide to Measure Theoretic Probability, vol. 8. Cambridge University Press, Cambridge (2002)

31. Rudin, W.: Real and Complex Analysis. Tata McGraw-Hill Education, London (1987)
32. Sampson, A., Panchekha, P., Mytkowicz, T., McKinley, K.S., Grossman, D., Ceze, L.: Expressing and verifying probabilistic assertions. ACM SIGPLAN Not. **49**(6), 112–122 (2014)
33. Smolka, S., Kumar, P., Foster, N., Kozen, D., Silva, A.: Cantor meets scott: Semantic foundations for probabilistic networks. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, pp. 557–571. ACM, New York (2017)
34. Staton, S.: Commutative semantics for probabilistic programming. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 855–879. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_32
35. Staton, S., Yang, H., Wood, F., Heunen, C., Kammar, O.: Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2016, pp. 525–534. ACM, New York (2016)
36. Wood, F., van de Meent, J., Mansinghka, V.: A new approach to probabilistic programming inference. CoRR, abs/1507.00996 (2015)

How long, O Bayesian network, will I sample thee?
A program analysis perspective on expected sampling times

Kevin Batz(✉), Benjamin Lucien Kaminski(✉), Joost-Pieter Katoen(✉), and Christoph Matheja(✉)

RWTH Aachen University, Aachen, Germany
kevin.batz@rwth-aachen.de,
{benjamin.kaminski,katoen,matheja}@cs.rwth-aachen.de

Abstract. Bayesian networks (BNs) are probabilistic graphical models for describing complex joint probability distributions. The main problem for BNs is inference: Determine the probability of an event given observed evidence. Since exact inference is often infeasible for large BNs, popular approximate inference methods rely on sampling.

We study the problem of determining the expected time to obtain a single valid sample from a BN. To this end, we translate the BN together with observations into a probabilistic program. We provide proof rules that yield the exact expected runtime of this program in a fully automated fashion. We implemented our approach and successfully analyzed various real–world BNs taken from the Bayesian network repository.

Keywords: Probabilistic programs · Expected runtimes
Weakest preconditions · Program verification

1 Introduction

Bayesian networks (BNs) are *probabilistic graphical models* representing joint probability distributions of sets of random variables with conditional dependencies. Graphical models are a popular and appealing modeling formalism, as they allow to succinctly represent complex distributions in a human–readable way. BNs have been intensively studied at least since 1985 [43] and have a wide range of applications including machine learning [24], speech recognition [50], sports betting [11], gene regulatory networks [18], diagnosis of diseases [27], and finance [39].

Probabilistic programs are programs with the key ability to draw values at random. Seminal papers by Kozen from the 1980s consider formal semantics [32] as well as initial work on verification [33,47]. McIver and Morgan [35] build on this work to further weakest–precondition style verification for imperative probabilistic programs.

The interest in probabilistic programs has been rapidly growing in recent years [20,23]. Part of the reason for this déjà vu is their use for representing probabilistic graphical models [31] such as BNs. The full potential of modern probabilistic programming languages like Anglican [48], Church [21], Figaro [44], R2 [40], or Tabular [22] is that they enable rapid prototyping and obviate the need to manually provide inference methods tailored to an individual model.

Probabilistic inference is the problem of determining the probability of an event given observed evidence. It is a major problem for both BNs and probabilistic programs, and has been subject to intense investigations by both theoreticians and practitioners for more than three decades; see [31] for a survey. In particular, it has been shown that for probabilistic programs exact inference is highly undecidable [28], while for BNs both *exact inference* as well as *approximate inference* to an arbitrary precision are NP–hard [12,13]. In light of these complexity–theoretical hurdles, a popular way to analyze probabilistic graphical models as well as probabilistic programs is to gather a large number of independent and identically distributed (i.i.d. for short) samples and then do statistical reasoning on these samples. In fact, all of the aforementioned probabilistic programming languages support sampling based inference methods.

Rejection sampling is a fundamental approach to obtain valid samples from BNs with observed evidence. In a nutshell, this method first samples from the joint (unconditional) distribution of the BN. If the sample complies with all evidence, it is valid and accepted; otherwise it is rejected and one has to resample.

Apart from rejection sampling, there are more sophisticated sampling techniques, which mainly fall in two categories: Markov Chain Monte Carlo (MCMC) and importance sampling. But while MCMC requires heavy hand–tuning and suffers from slow convergence rates on real–world instances [31, Chapter 12.3], virtually all variants of importance sampling rely again on rejection sampling [31,49].

A major problem with rejection sampling is that for poorly conditioned data, this approach might have to reject and resample very often in order to obtain just a single accepting sample. Even worse, being poorly conditioned need not be immediately evident for a given BN, let alone a probabilistic program. In fact, Gordon et al. [23, p. 177] point out that

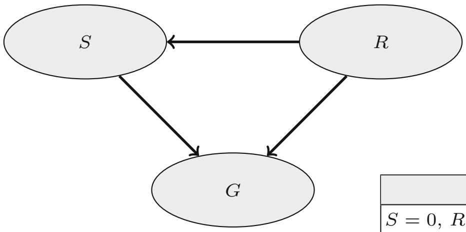
> "the main challenge in this setting [i.e. sampling based approaches] is that many samples that are generated during execution are ultimately rejected for not satisfying the observations."

If too many samples are rejected, the expected sampling time grows so large that sampling becomes infeasible. The expected sampling time of a BN is therefore a key figure for deciding whether sampling based inference is the method of choice.

How Long, O Bayesian Network, will I Sample Thee? More precisely, we use techniques from program verification to give an answer to the following question:

> Given a Bayesian network with observed evidence, how long does it take in expectation to obtain a *single* sample that satisfies the observations?

	$S = 0$	$S = 1$
$R = 0$	a	$1 - a$
$R = 1$	0.2	0.8



$R = 0$	$R = 1$
a	$1 - a$

	$G = 0$	$G = 1$
$S = 0,\ R = 0$	0.01	0.99
$S = 0,\ R = 1$	0.25	0.75
$S = 1,\ R = 0$	0.9	0.1
$S = 1,\ R = 1$	0.2	0.8

Fig. 1. A simple Bayesian network.

As an example, consider the BN in Fig. 1 which consists of just three nodes (random variables) that can each assume values 0 or 1. Each node X comes with a conditional probability table determining the probability of X assuming some value given the values of all nodes Y that X depends on (i.e. X has an incoming edge from Y), see [3, Appendix A.1] for detailed calculations. For instance, the probability that G assumes value 0, given that S and R are both assume 1, is 0.2. Note that this BN is paramterized by $a \in [0, 1]$.

Now, assume that our observed evidence is the event $G = 0$ and we apply rejection sampling to obtain *one* accepting sample from this BN. Then our approach will yield that a rejection sampling algorithm will, on average, require

$$\frac{200a^2 - 40a - 460}{89a^2 - 69a - 21}$$

guard evaluations, random assignments, etc. until it obtains a single sample that complies with the observation $G = 0$ (the underlying runtime model is discussed in detail in Sect. 3.3). By examination of this function, we see that for large ranges of values of a the BN is rather well–behaved: For $a \in [0.08,\ 0.78]$ the expected sampling time stays below 18. Above $a = 0.95$ the expected sampling time starts to grow rapidly up to 300.

While 300 is still moderate, we will see later that expected sampling times of real–world BNs can be much larger. For some BNs, the expected sampling time even exceeded 10^{18}, rendering sampling based methods infeasible. In this case, exact inference (despite NP–hardness) was a viable alternative (see Sect. 6).

Our Approach. We apply weakest precondition style reasoning a lá McIver and Morgan [35] and Kaminski et al. [30] to analyze both expected outcomes and *expected runtimes* (ERT) of a *syntactic fragment of* pGCL, which we call the *Bayesian Network Language* (BNL). Note that since BNL is a syntactic fragment of pGCL, every BNL program is a pGCL program but *not vice versa.* The main restriction of BNL is that (in contrast to pGCL) loops are of a special form that prohibits undesired data flow across multiple loop iterations. While this restriction renders BNL incapable of, for instance, counting the number of loop iterations[1], BNL is expressive enough to encode Bayesian networks with observed evidence.

For BNL, we develop dedicated proof rules to determine *exact* expected values and the *exact* ERT of any BNL program, including loops, without any user–supplied data, such as invariants [30,35], ranking or metering functions [19], (super)martingales [8–10], etc.

As a central notion behind these rules, we introduce *f–i.i.d.–ness* of probabilistic loops, a concept closely related to stochastic independence, that allows us to *rule out undesired parts of the data flow across loop iterations*. Furthermore, we show how every BN with observations is translated into a BNLprogram, such that

(a) executing the BNL program corresponds to sampling from the *conditional* joint distribution given by the BN and observed data, and
(b) the ERT of the BNL program corresponds to the expected time until a sample that satisfies the observations is obtained from the BN.

As a consequence, exact expected sampling times of BNs can be inferred by means of weakest precondition reasoning in a fully automated fashion. This can be seen as a first step towards formally evaluating the quality of a plethora of different sampling methods (cf. [31,49]) on source code level.

Contributions. To summarize, our main contributions are as follows:

- We develop easy–to–apply proof rules to reason about expected outcomes and expected runtimes of probabilistic programs with f–i.i.d. loops.
- We study a syntactic fragment of probabilistic programs, the Bayesian network language (BNL), and show that our proof rules are applicable to every BNL program; expected runtimes of BNL programs can thus be inferred.
- We give a formal translation from Bayesian networks with observations to BNL programs; expected sampling times of BNs can thus be inferred.
- We implemented a prototype tool that automatically analyzes the expected sampling time of BNs with observations. An experimental evaluation on real–world BNs demonstrates that very large expected sampling times (in the magnitude of millions of years) can be inferred within less than a second; This provides practitioners the means to decide whether sampling based methods are appropriate for their models.

Outline. We discuss related work in Sect. 2. Syntax and semantics of the probabilistic programming language pGCL are presented in Sect. 3. Our proof rules are introduced in Sect. 4 and applied to BNs in Sect. 5. Section 6 reports on experimental results and Sect. 7 concludes.

[1] An example of a program that is *not* expressible in BNL is given in Example 1.

2 Related Work

While various techniques for formal reasoning about runtimes and expected outcomes of probabilistic programs have been developed, e.g. [6,7,17,25,38], none of them explicitly apply formal methods to reason about Bayesian networks on source code level. In the following, we focus on approaches close to our work.

Weakest Preexpectation Calculus. Our approach builds upon the expected runtime calculus [30], which is itself based on work by Kozen [32,33] and McIver and Morgan [35]. In contrast to [30], we develop specialized proof rules for a clearly specified program fragment *without* requiring user–supplied invariants. Since finding invariants often requires heavy calculations, our proof rules contribute towards simplifying and automating verification of probabilistic programs.

Ranking Supermartingales. Reasoning about almost–sure termination is often based on ranking (super)martingales (cf. [8,10]). In particular, Chatterjee et al. [9] consider the class of affine probabilistic programs for which linear ranking supermartingales exist (Lrapp); thus proving (positive[2]) almost–sure termination for all programs within this class. They also present a doubly–exponential algorithm to approximate ERTs of Lrapp programs. While all BNL programs lie within Lrapp, our proof rules yield *exact* ERTs as *expectations* (thus allowing for compositional proofs), in contrast to a single number for a fixed initial state.

Bayesian Networks and Probabilistic Programs. Bayesian networks are a—if not the most—popular probabilistic graphical model (cf. [4,31] for details) for reasoning about conditional probabilities. They are closely tied to (a fragment of) probabilistic programs. For example, Infer.NET [36] performs inference by compiling a probabilistic program into a Bayesian network. While correspondences between probabilistic graphical models, such as BNs, have been considered in the literature [21,23,37], we are not aware of a formal soudness proof for a translation from classical BNs into probabilistic programs including conditioning.

Conversely, some probabilistic programming languages such as Church [21], Stan [26], and R2 [40] directly perform inference on the program level using sampling techniques similar to those developed for Bayesian networks. Our approach is a step towards understanding sampling based approaches formally: We obtain the exact expected runtime required to generate a sample that satisfies all observations. This may ultimately be used to evaluate the quality of a plethora of proposed sampling methods for Bayesian inference (cf. [31,49]).

3 Probabilistic Programs

We briefly present the probabilistic programming language that is used throughout this paper. Since our approach is embedded into weakest-precondition style approaches, we also recap calculi for reasoning about both expected outcomes and expected runtimes of probabilistic programs.

[2] Positive almost–sure termination means termination in finite expected time [5].

3.1 The Probabilistic Guarded Command Language

We enhance Dijkstra's Guarded Command Language [14,15] by a probabilistic construct, namely a random assignment. We thereby obtain a *probabilistic Guarded Command Language* (for a closely related language, see [35]).

Let Vars be a finite set of *program variables*. Moreover, let $\mathbb{Q}$ be the set of rational numbers, and let $\mathcal{D}(\mathbb{Q})$ be the set of discrete probability distributions over $\mathbb{Q}$. The set of *program states* is given by $\Sigma = \{\, \sigma \mid \sigma\colon \mathsf{Vars} \to \mathbb{Q} \,\}$.

A *distribution expression* μ is a function of type $\mu\colon \Sigma \to \mathcal{D}(\mathbb{Q})$ that takes a program state and maps it to a probability distribution on values from $\mathbb{Q}$. We denote by μ_σ the distribution obtained from applying σ to μ.

The probabilistic guarded command language (pGCL) is given by the grammar

$$
\begin{array}{lllr}
C & \longrightarrow & \texttt{skip} & \text{(effectless program)} \\
 & & \mid\ \texttt{diverge} & \text{(endless loop)} \\
 & & \mid\ x :\approx \mu & \text{(random assignment)} \\
 & & \mid\ C;\, C & \text{(sequential composition)} \\
 & & \mid\ \texttt{if}\,(\varphi)\,\{C\}\,\texttt{else}\,\{C\} & \text{(conditional choice)} \\
 & & \mid\ \texttt{while}\,(\varphi)\,\{C\} & \text{(while loop)} \\
 & & \mid\ \texttt{repeat}\,\{C\}\,\texttt{until}\,(\varphi)\ , & \text{(repeat–until loop)}
\end{array}
$$

where $x \in \mathsf{Vars}$ is a program variable, μ is a distribution expression, and φ is a Boolean expression guarding a choice or a loop. A pGCL program that contains neither `diverge`, nor `while`, nor `repeat` – `until` loops is called loop–free.

For $\sigma \in \Sigma$ and an arithmetical expression E over Vars, we denote by $\sigma(E)$ the evaluation of E in σ, i.e. the value that is obtained by evaluating E after replacing any occurrence of any program variable x in E by the value $\sigma(x)$. Analogously, we denote by $\sigma(\varphi)$ the evaluation of a guard φ in state σ to either true or false. Furthermore, for a value $v \in \mathbb{Q}$ we write $\sigma[x \mapsto v]$ to indicate that we set program variable x to value v in program state σ, i.e.[3]

$$
\sigma[x \mapsto v] \;=\; \lambda\, y \bullet \begin{cases} v, & \text{if } y = x \\ \sigma(y), & \text{if } y \neq x\ . \end{cases}
$$

We use the Iverson bracket notation to associate with each guard its according indicator function. Formally, the Iverson bracket $[\varphi]$ of φ is thus defined as the function $[\varphi] = \lambda\,\sigma \bullet\ \sigma(\varphi)$.

Let us briefly go over the pGCL constructs and their effects: `skip` does not alter the current program state. The program `diverge` is an infinite busy loop, thus takes infinite time to execute. It returns no final state whatsoever.

The random assignment $x :\approx \mu$ is (a) the only construct that can actually alter the program state and (b) the only construct that may introduce random

[3] We use λ–expressions to construct functions: Function $\lambda X \bullet\ \epsilon$ applied to an argument α evaluates to ϵ in which every occurrence of X is replaced by α.

behavior into the computation. It takes the current program state σ, then *samples* a value v from probability distribution μ_σ, and then assigns v to program variable x. An example of a random assignment is

$$x :\approx {}^1\!/_2 \cdot \langle 5 \rangle + {}^1\!/_6 \cdot \langle y+1 \rangle + {}^1\!/_3 \cdot \langle y-1 \rangle \ .$$

If the current program state is σ, then the program state is altered to either $\sigma\left[x \mapsto 5\right]$ with probability $1/2$, or to $\sigma\left[x \mapsto \sigma(y)+1\right]$ with probability $1/6$, or to $\sigma\left[x \mapsto \sigma(y)-1\right]$ with probability $1/3$. The remainder of the pGCL constructs are standard programming language constructs.

In general, a pGCL program C is executed on an input state and yields a *probability distribution* over final states due to possibly occurring random assignments inside of C. We denote that resulting distribution by $\llbracket C \rrbracket_\sigma$. Strictly speaking, programs can yield *subdistributions*, i.e. probability distributions whose total mass may be below 1. The “missing”probability mass represents the probability of nontermination. Let us conclude our presentation of pGCLwith an example:

Example 1 (Geometric Loop). Consider the program C_{geo} given by

$$\begin{aligned}&x :\approx 0; \quad c :\approx {}^1\!/_2 \cdot \langle 0 \rangle + {}^1\!/_2 \cdot \langle 1 \rangle; \\ &\texttt{while}\,(c=1)\,\{x :\approx x+1;\ c :\approx {}^1\!/_2 \cdot \langle 0 \rangle + {}^1\!/_2 \cdot \langle 1 \rangle\}\end{aligned}$$

This program basically keeps flipping coins until it flips, say, heads ($c = 0$). In x it counts the number of unsuccessful trials.[4] In effect, it almost surely sets c to 0 and moreover it establishes a geometric distribution on x. The resulting distribution is given by

$$\llbracket C_{geo} \rrbracket_\sigma (\tau) \;=\; \sum_{n=0}^{\omega} \left[\tau = \sigma\left[c, x \mapsto 0, n\right]\right] \cdot \frac{1}{2^{n+1}} \ . \qquad \triangle$$

3.2 The Weakest Preexpectation Transformer

We now present the weakest preexpectation transformer wp for reasoning about expected outcomes of executing probabilistic programs in the style of McIver and Morgan [35]. Given a random variable f mapping program states to reals, it allows us to reason about the expected value of f after executing a probabilistic program on a given state.

Expectations. The random variables the wp transformer acts upon are taken from a set of so-called expectations, a term coined by McIver and Morgan [35]:

[4] This counting is also the reason that C_{geo} is an example of a program that is not expressible in our BNL language that we present later.

Definition 1 (Expectations). *The set of expectations $\mathbb{E}$ is defined as*

$$\mathbb{E} \;=\; \left\{ f \;\middle|\; f\colon \Sigma \to \mathbb{R}_{\geq 0}^{\infty} \right\} \;.$$

We will use the notation $f[x/E]$ to indicate the replacement *of every occurrence of x in f by E. Since x, however, does not actually occur in f, we more formally define $f[x/E] = \lambda\sigma \bullet\; f(\sigma\,[x \mapsto \sigma(E)])$.*

A complete partial order $\leq$ on $\mathbb{E}$ is obtained by point–wise lifting the canonical total order on $\mathbb{R}_{\geq 0}^{\infty}$, i.e.

$$f_1 \;\preceq\; f_2 \quad \textit{iff} \quad \forall \sigma \in \Sigma\colon \quad f_1(\sigma) \;\leq\; f_2(\sigma) \;.$$

Its least element is given by $\lambda\sigma \bullet\; 0$ which we (by slight abuse of notation) also denote by 0. Suprema are constructed pointwise, i.e. for $S \subseteq \mathbb{E}$ the supremum $\sup S$ *is given by* $\sup S = \lambda\sigma \bullet\; \sup_{f \in S}\; f(\sigma)$.

We allow expectations to map only to positive reals, so that we have a complete partial order readily available, which would not be the case for expectations of type $\Sigma \to \mathbb{R} \cup \{-\infty,\, +\infty\}$. A wp calculus that *can* handle expectations of such type needs more technical machinery and cannot make use of this underlying natural partial order [29]. Since we want to reason about ERTs which are by nature non–negative, we will not need such complicated calculi.

Notice that we use a slightly different definition of expectations than McIver and Morgan [35], as we allow for *unbounded* expectations, whereas [35] requires that expectations are *bounded*. This however would prevent us from capturing ERTs, which are potentially unbounded.

Expectation Transformers. For reasoning about the expected value of $f \in \mathbb{E}$ after execution of C, we employ a backward–moving weakest preexpectation transformer $\mathsf{wp}[\![C]\!]\colon \mathbb{E} \to \mathbb{E}$, that maps a *postexpectation* $f \in \mathbb{E}$ to a *preexpectation* $\mathsf{wp}\,[\![C]\!]\,(f) \in \mathbb{E}$, such that $\mathsf{wp}\,[\![C]\!]\,(f)\,(\sigma)$ is the expected value of f after executing C on initial state σ. Formally, if C executed on input σ yields final distribution $[\![C]\!]_\sigma$, then the *weakest preexpectation* $\mathsf{wp}\,[\![C]\!]\,(f)$ *of C with respect to postexpectation f is given by*

$$\mathsf{wp}\,[\![C]\!]\,(f)\,(\sigma) \;=\; \int_{\Sigma} f \; d[\![C]\!]_\sigma \;, \tag{1}$$

where we denote by $\int_A h\, d\nu$ the expected value of a random variable $h\colon A \to \mathbb{R}_{\geq 0}^{\infty}$ with respect to a probability distribution $\nu\colon A \to [0,\,1]$. Weakest preexpectations can be defined in a very systematic way:

Definition 2 (The wp Transformer [35]). *The weakest preexpectation transformer* $\mathsf{wp}\colon \mathsf{pGCL} \to \mathbb{E} \to \mathbb{E}$ *is defined by induction on all* pGCL *programs according to the rules in Table 1. We call $F_f(X) = [\neg\varphi] \cdot f + [\varphi] \cdot \mathsf{wp}\,[\![C]\!]\,(X)$ the* wp*–*characteristic functional *of the loop* `while` $(\varphi)\,\{C\}$ *with respect to postexpectation f. For a given* wp*–characteristic function F_f, we call the sequence $\{F_f^n(0)\}_{n \in \mathbb{N}}$ the* orbit of F_f.

Table 1. Rules for the wp–transformer.

C	$\mathsf{wp}\,[\![C]\!]\,(f)$
`skip`	f
`diverge`	0
$x :\approx \mu$	$\lambda\sigma\bullet\ \int_{\mathbb{Q}} \big(\lambda v\bullet\ f[x/v]\big)\ d\mu_\sigma$
$\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$	$[\varphi]\cdot\mathsf{wp}\,[\![C_1]\!]\,(f) + [\neg\varphi]\cdot\mathsf{wp}\,[\![C_2]\!]\,(f)$
$C_1;\ C_2$	$\mathsf{wp}\,[\![C_1]\!]\,(\mathsf{wp}\,[\![C_2]\!]\,(f))$
$\texttt{while}\,(\varphi)\,\{C'\}$	$\mathsf{lfp}\ X\bullet\ \ [\neg\varphi]\cdot f + [\varphi]\cdot\mathsf{wp}\,[\![C']\!]\,(X)$
$\texttt{repeat}\,\{C'\}\,\texttt{until}\,(\varphi)$	$\mathsf{wp}\,[\![C';\ \texttt{while}\,(\neg\varphi)\,\{C'\}]\!]\,(f)$

Let us briefly go over the definitions in Table 1: For `skip` the program state is not altered and thus the expected value of f is just f. The program `diverge` will never yield any final state. The distribution over the final states yielded by `diverge` is thus the null distribution $\nu_0(\tau) = 0$, that assigns probability 0 to *every* state. Consequently, the expected value of f after execution of `diverge` is given by $\int_\Sigma\ f\ d\nu_0 = \sum_{\tau\in\Sigma} 0\cdot f(\tau) = 0$.

The rule for the random assignment $x :\approx \mu$ is a bit more technical: Let the current program state be σ. Then for every value $v \in \mathbb{Q}$, the random assignment assigns v to x with probability $\mu_\sigma(v)$, where σ is the current program state. The value of f after assigning v to x is $f(\sigma\,[x \mapsto v]) = f[x/v](\sigma)$ and therefore the expected value of f after executing the random assignment is given by

$$\sum_{v\in\mathbb{Q}} \mu_\sigma(v)\cdot f[x/v](\sigma) \ = \ \int_{\mathbb{Q}} \big(\lambda v\bullet\ f[x/v](\sigma)\big)\ d\mu_\sigma\ .$$

Expressed as a function of σ, the latter yields precisely the definition in Table 1.

The definition for the conditional choice $\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$ is not surprising: if the current state satisfies φ, we have to opt for the weakest preexpectation of C_1, whereas if it does not satisfy φ, we have to choose the weakest preexpectation of C_2. This yields precisely the definition in Table 1.

The definition for the sequential composition $C_1;\ C_2$ is also straightforward: We first determine $\mathsf{wp}\,[\![C_2]\!]\,(f)$ to obtain the expected value of f after executing C_2. Then we mentally prepend the program C_2 by C_1 and therefore determine the expected value of $\mathsf{wp}\,[\![C_2]\!]\,(f)$ after executing C_1. This gives the weakest preexpectation of $C_1;\ C_2$ with respect to postexpectation f.

The definition for the while loop makes use of a least fixed point, which is a standard construction in program semantics. Intuitively, the fixed point iteration of the wp–characteristic functional, given by 0, $F_f(0)$, $F_f^2(0)$, $F_f^3(0)$, ..., corresponds to the portion the expected value of f after termination of the loop, that can be collected within at most 0, 1, 2, 3, ... loop guard evaluations.

The Kleene Fixed Point Theorem [34] ensures that this iteration converges to the least fixed point, i.e.

$$\sup_{n \in \mathbb{N}} F_f^n(0) \;=\; \mathsf{lfp}\ F_f \;=\; \mathsf{wp}\,[\![\mathtt{while}\,(\varphi)\,\{C\}]\!]\,(f)\,.$$

By inspection of the above equality, we see that the least fixed point is exactly the construct that we want for while loops, since $\sup_{n \in \mathbb{N}} F_f^n(0)$ in principle allows the loop to run for any number of iterations, which captures precisely the semantics of a while loop, where the number of loop iterations is—in contrast to e.g. `for` loops—not determined upfront.

Finally, since $\mathtt{repeat}\,\{C\}\,\mathtt{until}\,(\varphi)$ is syntactic sugar for C; $\mathtt{while}\,(\varphi)\,\{C\}$, we simply define the weakest preexpectation of the former as the weakest preexpectation of the latter. Let us conclude our study of the effects of the wp transformer by means of an example:

Example 2. Consider the following program C:

$$c :\approx {}^1\!/_3 \cdot \langle 0 \rangle + {}^2\!/_3 \cdot \langle 1 \rangle;$$
$$\mathtt{if}\,(c = 0)\,\{x :\approx {}^1\!/_2 \cdot \langle 5 \rangle + {}^1\!/_6 \cdot \langle y+1 \rangle + {}^1\!/_3 \cdot \langle y-1 \rangle\}\,\mathtt{else}\,\{\mathtt{skip}\}$$

Say we wish to reason about the expected value of $x + c$ after execution of the above program. We can do so by calculating $\mathsf{wp}\,[\![C]\!]\,(x+c)$ using the rules in Table 1. This calculation in the end yields $\mathsf{wp}\,[\![C]\!]\,(x+c) \;=\; {}^{3y+26}\!/_{18}$ The expected valuation of the expression $x + c$ after executing C is thus ${}^{3y+26}\!/_{18}$. Note that $x + c$ can be thought of as an expression that is evaluated in the final states after execution, whereas ${}^{3y+26}\!/_{18}$ must be evaluated in the initial state before execution of C. △

Healthiness Conditions of wp. The wp transformer enjoys some useful properties, sometimes called *healthiness conditions* [35]. Two of these healthiness conditions that we will heavily make use of are given below:

Theorem 1 (Healthiness Conditions for the wp Transformer [35]). *For all $C \in \mathsf{pGCL}$, $f_1, f_2 \in \mathbb{E}$, and $a \in \mathbb{R}_{\geq 0}$, the following holds:*

1. $\mathsf{wp}\,[\![C]\!]\,(a \cdot f_1 + f_2) \;=\; a \cdot \mathsf{wp}\,[\![C]\!]\,(f_1) + \mathsf{wp}\,[\![C]\!]\,(f_2)$ *(linearity)*
2. $\mathsf{wp}\,[\![C]\!]\,(0) \;=\; 0$ *(strictness).*

3.3 The Expected Runtime Transformer

While for deterministic programs we can speak of *the* runtime of a program on a given input, the situation is different for probabilistic programs: For those we instead have to speak of the *expected runtime* (ERT). Notice that the ERT can be finite (even constant) while the program may still admit infinite executions. An example of this is the geometric loop in Example 1.

A wp–like transformer designed specifically for reasoning about ERTs is the ert transformer [30]. Like wp, it is of type $\mathsf{ert}[\![C]\!] : \mathbb{E} \rightarrow \mathbb{E}$ and it can be shown that

Table 2. Rules for the ert–transformer.

C	$\mathsf{ert}\,[\![C]\!]\,(f)$
$\texttt{skip}$	$1+f$
$\texttt{diverge}$	∞
$x :\approx \mu$	$1+\lambda\sigma\bullet\ \int_{\mathbb{Q}} \big(\lambda v\bullet\ f[x/v]\big)\ d\mu_\sigma$
$\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$	$1+[\varphi]\cdot\mathsf{ert}\,[\![C_1]\!]\,(f)+[\neg\varphi]\cdot\mathsf{ert}\,[\![C_2]\!]\,(f)$
$C_1;\,C_2$	$\mathsf{ert}\,[\![C_1]\!]\,\big(\big(\mathsf{ert}\,[\![C_2]\!]\,(f)\,\big)\big)$
$\texttt{while}\,(\varphi)\,\{C'\}$	$\mathsf{lfp}\ X\bullet\ 1+[\neg\varphi]\cdot f+[\varphi]\cdot\mathsf{ert}\,[\![C']\!]\,(X)$
$\texttt{repeat}\,\{C'\}\,\texttt{until}\,(\varphi)$	$\mathsf{ert}\,[\![C';\,\texttt{while}\,(\neg\varphi)\,\{C'\}]\!]\,(f)$

$\mathsf{ert}\,[\![C]\!]\,(0)\,(\sigma)$ is precisely the *expected runtime of executing C on input σ*. More generally, if $f\colon \Sigma\to\mathbb{R}_{\geq 0}^{\infty}$ measures the time that is needed after executing C (thus f is evaluated in the final states after termination of C), then $\mathsf{ert}\,[\![C]\!]\,(f)\,(\sigma)$ is the expected time that is needed to run C on input σ and then let time f pass. For a more in–depth treatment of the ert transformer, see [30, Sect. 3]. The transformer is defined as follows:

Definition 3 (The ert Transformer [30]). *The expected runtime transformer* $\mathsf{ert}\colon \mathsf{pGCL}\to\mathbb{E}\to\mathbb{E}$ *is defined by induction on all* pGCL *programs according to the rules given in Table 2. We call $F_f(X)=1+[\neg\varphi]\cdot f+[\varphi]\cdot\mathsf{wp}\,[\![C]\!]\,(X)$ the* ert–characteristic functional *of the loop* $\texttt{while}\,(\varphi)\,\{C\}$ *with respect to postexpectation f. As with* wp*, for a given* ert*–characteristic function F_f, we call the sequence $\{F_f^n(0)\}_{n\in\mathbb{N}}$ the* orbit *of F_f. Notice that*

$$\mathsf{ert}\,[\![\texttt{while}\,(\varphi)\,\{C\}]\!]\,(f) \;=\; \mathsf{lfp}\ F_f \;=\; \sup\ \{F_f^n(0)\}_{n\in\mathbb{N}}.$$

The rules for ert are very similar to the rules for wp. The runtime model we assume is that $\texttt{skip}$ statements, random assignments, and guard evaluations for both conditional choice and while loops cost one unit of time. This runtime model can easily be adopted to count only the number of loop iterations or only the number of random assignments, etc. We conclude with a strong connection between the wp and the ert transformer, that is crucial in our proofs:

Theorem 2 (Decomposition of ert [41]). *For any $C\in\mathsf{pGCL}$ and $f\in\mathbb{E}$,*

$$\mathsf{ert}\,[\![C]\!]\,(f) \;=\; \mathsf{ert}\,[\![C]\!]\,(0)+\mathsf{wp}\,[\![C]\!]\,(f)\,.$$

4 Expected Runtimes of i.i.d Loops

We derive a proof rule that allows to determine *exact ERTs of independent and identically distributed loops* (or *i.i.d. loops* for short). Intuitively, a loop

```
while ((x − 5)^2 + (y − 5)^2 ≥ 25) {
    x :≈ Unif[0 ... 10];
    y :≈ Unif[0 ... 10]
}
```

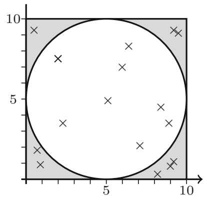


Fig. 2. An i.i.d. loop sampling a point within a circle uniformly at random using rejection sampling. The picture on the right–hand side visualizes the procedure: In each iteration a point ($\times$) is sampled. If we obtain a point within the white area inside the square, we terminate. Otherwise, i.e. if we obtain a point within the gray area outside the circle, we resample.

is i.i.d. if the distributions of states that are reached at the end of different loop iterations are equal. This is the case whenever there is no data flow across different iterations. In the non–probabilistic case, such loops either terminate after exactly one iteration or never. This is different for probabilistic programs.

As a running example, consider the program C_{circle} in Fig. 2. C_{circle} samples a point within a circle with center $(5, 5)$ and radius $r = 5$ uniformly at random using rejection sampling. In each iteration, it samples a point $(x, y) \in [0, \ldots, 10]^2$ within the square (with some fixed precision). The loop ensures that we resample if a sample is not located within the circle. Our proof rule will allow us to systematically determine the ERT of this loop, i.e. the average amount of time required until a single point within the circle is sampled.

Towards obtaining such a proof rule, we first present a syntactical notion of the i.i.d. property. It relies on expectations that are not affected by a pGCL program:

Definition 4. *Let $C \in$ pGCL and $f \in \mathbb{E}$. Moreover, let $\mathsf{Mod}(C)$ denote the set of all variables that occur on the left–hand side of an assignment in C, and let $\mathsf{Vars}(f)$ be the set of all variables that "occur in f", i.e. formally*

$$x \in \mathsf{Vars}(f) \qquad \text{iff} \qquad \exists\, \sigma\ \exists\, v, v' \colon \quad f(\sigma\,[x \mapsto v]) \;\neq\; f(\sigma\,[x \mapsto v']).$$

Then f is unaffected *by C, denoted $f \not\Vdash C$, iff $\mathsf{Vars}(f) \cap \mathsf{Mod}(C) = \emptyset$.*

We are interested in expectations that are unaffected by pGCL programs because of a simple, yet useful observation: If $g \not\Vdash C$, then g *can be treated like a constant* w.r.t. the transformer wp (i.e. like the a in Theorem 1 (1)). For our running example C_{circle} (see Fig. 2), the expectation $f = \mathsf{wp}\,[\![C_{body}]\!]\,([x + y \leq 10])$ is unaffected by the loop body C_{body} of C_{circle}. Consequently, we have $\mathsf{wp}\,[\![C_{body}]\!]\,(f) = f \cdot \mathsf{wp}\,[\![C_{body}]\!]\,(1) = f$. In general, we obtain the following property:

Lemma 1 (Scaling by Unaffected Expectations). *Let $C \in$ pGCL and $f, g \in \mathbb{E}$. Then $g \not\Vdash C$ implies $\mathsf{wp}\,[\![C]\!]\,(g \cdot f) = g \cdot \mathsf{wp}\,[\![C]\!]\,(f)$.*

Proof. By induction on the structure of C. See [3, Appendix A.2]. □

We develop a proof rule that only requires that both the probability of the guard evaluating to true after one iteration of the loop body (i.e. $\mathsf{wp}\,[\![C]\!]\,([\varphi])$) as well as the expected value of $[\neg\varphi]\cdot f$ after one iteration (i.e. $\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)$) are unaffected by the loop body. We thus define the following:

Definition 5 (f–Independent and Identically Distributed Loops). *Let $C \in \mathsf{pGCL}$, φ be a guard, and $f \in \mathbb{E}$. Then we call the loop* $\texttt{while}\,(\varphi)\,\{C\}$ *f–*independent and identically distributed *(or f–*i.i.d. *for short), if both*

$$\mathsf{wp}\,[\![C]\!]\,([\varphi]) \not\Vdash C \qquad \textit{and} \qquad \mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f) \not\Vdash C.$$

Example 3. Our example program C_{circle} (see Fig. 2) is f–i.i.d. for all $f \in \mathbb{E}$. This is due to the fact that

$$\mathsf{wp}\,[\![C_{body}]\!]\,\big([(x-5)^2+(y-5)^2 \geq 25]\big) \;=\; \frac{48}{121} \not\Vdash C_{body} \qquad \text{(by Table 1)}$$

and (again for some fixed precision $p \in \mathbb{N}\setminus\{0\}$)

$$\begin{aligned}&\mathsf{wp}\,[\![C_{body}]\!]\,\big([(x-5)^2+(y-5)^2 > 25]\cdot f\big)\\ &= \frac{1}{121}\cdot\sum_{i=0}^{10p}\sum_{j=0}^{10p}\big[({}^{i}\!/\!{}_{p}-5)^2+({}^{j}\!/\!{}_{p}-5)^2 > 25\big]\cdot f[x/({}^{i}\!/\!{}_{p}), y/({}^{j}\!/\!{}_{p})] \not\Vdash C_{body}. \quad \triangle\end{aligned}$$

Our main technical Lemma is that we can express the orbit of the wp–characteristic function as a partial geometric series:

Lemma 2 (Orbits of f–i.i.d. Loops). *Let $C \in \mathsf{pGCL}$, φ be a guard, $f \in \mathbb{E}$ such that the loop* $\texttt{while}\,(\varphi)\,\{C\}$ *is f–i.i.d, and let F_f be the corresponding* wp*–characteristic function. Then for all $n \in \mathbb{N}\setminus\{0\}$, it holds that*

$$F_f^n(0) \;=\; [\varphi]\cdot\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)\cdot\sum_{i=0}^{n-2}\Big(\mathsf{wp}\,[\![C]\!]\,([\varphi])^i\Big) \;+\; [\neg\varphi]\cdot f.$$

Proof. By use of Lemma 1, see [3, Appendix A.3].

Using this precise description of the wp orbits, we now establish proof rules for f–i.i.d. loops, first for wp and later for ert.

Theorem 3 (Weakest Preexpectations of f–i.i.d. Loops). *Let $C \in \mathsf{pGCL}$, φ be a guard, and $f \in \mathbb{E}$. If the loop* $\texttt{while}\,(\varphi)\,\{C\}$ *is f–i.i.d., then*

$$\mathsf{wp}\,[\![\texttt{while}\,(\varphi)\,\{C\}]\!]\,(f) \;=\; [\varphi]\cdot\frac{\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)}{1-\mathsf{wp}\,[\![C]\!]\,([\varphi])} + [\neg\varphi]\cdot f\ ,$$

where we define $\frac{0}{0} := 0$.

Proof. We have

$$\begin{aligned}
&\mathsf{wp}\,[\![\texttt{while}\,(\varphi)\,\{C\}]\!]\,(f)\\
&= \sup_{n\in\mathbb{N}}\ F_f^n(0) && \text{(by Definition 2)}\\
&= \sup_{n\in\mathbb{N}}\ [\varphi]\cdot\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)\cdot\sum_{i=0}^{n-2}\left(\mathsf{wp}\,[\![C]\!]\,([\varphi])^i\right)+[\neg\varphi]\cdot f && \text{(by Lemma 2)}\\
&= [\varphi]\cdot\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)\cdot\sum_{i=0}^{\omega}\left(\mathsf{wp}\,[\![C]\!]\,([\varphi])^i\right)+[\neg\varphi]\cdot f. && (\dagger)
\end{aligned}$$

The preexpectation (†) is to be evaluated in some state σ for which we have two cases: The first case is when $\mathsf{wp}\,[\![C]\!]\,([\varphi])\,(\sigma) < 1$. Using the closed form of the geometric series, i.e. $\sum_{i=0}^{\omega} q = \frac{1}{1-q}$ if $|q| < 1$, we get

$$\begin{aligned}
&[\varphi]\,(\sigma)\cdot\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)\,(\sigma)\cdot\sum_{i=0}^{\omega}\left(\mathsf{wp}\,[\![C]\!]\,([\varphi])\,(\sigma)^i\right)+[\neg\varphi]\,(\sigma)\cdot f(\sigma)\\
&\hfill (\dagger\text{ instantiated in }\sigma)\\
&= [\varphi]\,(\sigma)\cdot\frac{\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)\,(\sigma)}{1-\mathsf{wp}\,[\![C]\!]\,([\varphi])\,(\sigma)}+[\neg\varphi]\,(\sigma)\cdot f(\sigma).\\
&\hfill \text{(closed form of geometric series)}
\end{aligned}$$

The second case is when $\mathsf{wp}\,[\![C]\!]\,([\varphi])\,(\sigma) = 1$. This case is technically slightly more involved. The full proof can be found in [3, Appendix A.4]. □

We now derive a similar proof rule for the ERT of an f–i.i.d. loop $\texttt{while}\,(\varphi)\,\{C\}$.

Theorem 4 (Proof Rule for ERTs of *f*–i.i.d. Loops). *Let $C \in \mathsf{pGCL}$, φ be a guard, and $f \in \mathbb{E}$ such that all of the following conditions hold:*

1. $\texttt{while}\,(\varphi)\,\{C\}$ *is f–i.i.d.*
2. $\mathsf{wp}\,[\![C]\!]\,(1) = 1$ *(loop body terminates almost–surely).*
3. $\mathsf{ert}\,[\![C]\!]\,(0) \not\Cap C$ *(every iteration runs in the same expected time).*

Then for the ERT of the loop $\texttt{while}\,(\varphi)\,\{C\}$ w.r.t. postruntime f it holds that

$$\mathsf{ert}\,[\![\texttt{while}\,(\varphi)\,\{C\}]\!]\,(f) \;=\; 1+\frac{[\varphi]\cdot(1+\mathsf{ert}\,[\![C]\!]\,([\neg\varphi]\cdot f))}{1-\mathsf{wp}\,[\![C]\!]\,([\varphi])}+[\neg\varphi]\cdot f\ ,$$

where we define $\frac{0}{0} \coloneqq 0$ and $\frac{a}{0} \coloneqq \infty$, for $a \neq 0$.

Proof. We first prove

$$\mathsf{ert}\,[\![\texttt{while}\,(\varphi)\,\{C\}]\!]\,(0) \;=\; 1+[\varphi]\cdot\frac{1+\mathsf{ert}\,[\![C]\!]\,(0)}{1-\mathsf{wp}\,[\![C]\!]\,([\varphi])}. \qquad (\ddagger)$$

To this end, we propose the following expression as the orbit of the ert–characteristic function of the loop w.r.t. 0:

$$F_0^n(0) \;=\; 1 + [\varphi] \cdot \left(\mathsf{ert}\,[\![C]\!]\,(0) \cdot \sum_{i=0}^{n} \mathsf{wp}\,[\![C]\!]\,([\varphi])^i \;+\; \sum_{i=0}^{n-1} \mathsf{wp}\,[\![C]\!]\,([\varphi])^i \right)$$

For a verification that the above expression is indeed the correct orbit, we refer to the rigorous proof of this theorem in [3, Appendix A.5]. Now, analogously to the reasoning in the proof of Theorem 3 (i.e. using the closed form of the geometric series and case distinction on whether $\mathsf{wp}\,[\![C]\!]\,([\varphi]) < 1$ or $\mathsf{wp}\,[\![C]\!]\,([\varphi]) = 1$), we get that the supremum of this orbit is indeed the right–hand side of (‡). To complete the proof, consider the following:

$$\begin{aligned}
&\mathsf{ert}\,[\![\texttt{while}\,(\varphi)\,\{C\}]\!]\,(f) \\
&= \mathsf{ert}\,[\![\texttt{while}\,(\varphi)\,\{C\}]\!]\,(0) + \mathsf{wp}\,[\![\texttt{while}\,(\varphi)\,\{C\}]\!]\,(f) && \text{(by Theorem 2)} \\
&= 1 + [\varphi] \cdot \frac{1 + \mathsf{ert}\,[\![C]\!]\,(0)}{1 - \mathsf{wp}\,[\![C]\!]\,([\varphi])} + [\varphi] \cdot \frac{\mathsf{wp}\,[\![C]\!]\,([\neg\varphi] \cdot f)}{1 - \mathsf{wp}\,[\![C]\!]\,([\varphi])} + [\neg\varphi] \cdot f \\
& && \text{(by (‡) and Theorem 3)} \\
&= 1 + [\varphi] \cdot \frac{1 + \mathsf{ert}\,[\![C]\!]\,([\neg\varphi] \cdot f)}{1 - \mathsf{wp}\,[\![C]\!]\,([\varphi])} + [\neg\varphi] \cdot f && \text{(by Theorem 2)}
\end{aligned}$$

□

5 A Programming Language for Bayesian Networks

So far we have derived proof rules for formal reasoning about expected outcomes and expected run-times of i.i.d. loops (Theorems 3 and 4). In this section, we apply these results to develop a syntactic pGCL fragment that allows exact computations of closed forms of ERTs. In particular, no invariants, (super)martingales or fixed point computations are required.

After that, we show how BNs with observations can be translated into pGCL programs within this fragment. Consequently, we call our pGCL fragment the *Bayesian Network Language*. As a result of the above translation, we obtain a systematic and automatable approach to compute the *expected sampling time* of a BN in the presence of observations. That is, the expected time it takes to obtain a single sample that satisfies all observations.

5.1 The Bayesian Network Language

Programs in the Bayesian Network Language are organized as sequences of blocks. Every block is associated with a single variable, say x, and satisfies two constraints: First, no variable other than x is modified inside the block, i.e. occurs on the left–hand side of a random assignment. Second, every variable accessed inside of a guard has been initialized before. These restrictions ensure that there is no data flow across multiple executions of the same block. Thus, intuitively, all loops whose body is composed from blocks (as described above) are f–i.i.d. loops.

Definition 6 (The Bayesian Network Language). *Let* $\mathsf{Vars} = \{x_1, x_2, \ldots\}$ *be a finite set of program variables as in Sect. 3. The set of programs in Bayesian Network Language, denoted* BNL, *is given by the grammar*

$$\begin{aligned} C &\longrightarrow Seq \quad | \quad \texttt{repeat}\,\{Seq\}\,\texttt{until}\,(\psi) \quad | \quad C;\, C \\ Seq &\longrightarrow Seq;\, Seq \quad | \quad B_{x_1} \quad | \quad B_{x_2} \quad | \quad \ldots \\ B_{x_i} &\longrightarrow x_i :\approx \mu \quad | \quad \texttt{if}\,(\varphi)\,\{x_i :\approx \mu\}\,\texttt{else}\,\{B_{x_i}\} \\ & \qquad\qquad\qquad\qquad\qquad\qquad \text{(rule exists for all } x_i \in \mathsf{Vars}) \end{aligned}$$

where $x_i \in \mathsf{Vars}$ *is a program variable, all variables in* φ *have been initialized before, and* B_{x_i} *is a non–terminal parameterized with program variable* $x_i \in \mathsf{Vars}$. *That is, for all* $x_i \in \mathsf{Vars}$ *there is a non–terminal* B_{x_i}. *Moreover,* ψ *is an arbitrary guard and* μ *is a distribution expression of the form* $\mu = \sum_{j=1}^{n} p_j \cdot \langle a_j \rangle$ *with* $a_j \in \mathbb{Q}$ *for* $1 \leq j \leq n$.

Example 4. Consider the BNL program C_{dice}:

$$x_1 :\approx \texttt{Unif}[1\ldots6];\ \texttt{repeat}\,\{x_2 :\approx \texttt{Unif}[1\ldots6]\}\,\texttt{until}\,(x_2 \geq x_1)$$

This program first throws a fair die. After that it keeps throwing a second die until its result is at least as large as the first die. △

For any $C \in \mathsf{BNL}$, our goal is to compute the exact ERT of C, i.e. $\mathsf{ert}\,[\![C]\!]\,(0)$. In case of loop–free programs, this amounts to a straightforward application of the ert calculus presented in Sect. 3. To deal with loops, however, we have to perform fixed point computations or require user–supplied artifacts, e.g. invariants, supermartingales, etc. For BNL programs, on the other hand, it suffices to apply the proof rules developed in Sect. 4. As a result, we directly obtain an exact closed form solution for the ERT of a loop. This is a consequence of the fact that all loops in BNL are f–i.i.d., which we establish in the following.

By definition, every loop in BNL is of the form $\texttt{repeat}\,\{B_{x_i}\}\,\texttt{until}\,(\psi)$, which is equivalent to $B_{x_i};\ \texttt{while}\,(\neg\psi)\,\{B_{x_i}\}$. Hence, we want to apply Theorem 4 to that while loop. Our first step is to discharge the theorem's premises:

Lemma 3. *Let Seq be a sequence of* BNL*–blocks,* $g \in \mathbb{E}$*, and* ψ *be a guard. Then:*

1. *The expected value of* g *after executing Seq is unaffected by Seq. That is,* $\mathsf{wp}\,[\![Seq]\!]\,(g) \not\Cap Seq$.
2. *The ERT of Seq is unaffected by Seq, i.e.* $\mathsf{ert}\,[\![Seq]\!]\,(0) \not\Cap Seq$.
3. *For every* $f \in \mathbb{E}$*, the loop* $\texttt{while}\,(\neg\psi)\,\{Seq\}$ *is* f*–i.i.d.*

Proof. 1. is proven by induction on the length of the sequence of blocks *Seq* and 2. is a consequence of 1., see [3, Appendix A.6]. 3. follows immediately from 1. by instantiating g with $[\neg\psi]$ and $[\psi] \cdot f$, respectively. □

We are now in a position to derive a closed form for the ERT of loops in BNL.

Theorem 5. *For every loop* $\texttt{repeat}\,\{\mathit{Seq}\}\,\texttt{until}\,(\psi) \in \mathsf{BNL}$ *and every* $f \in \mathbb{E}$*,*

$$\mathsf{ert}\,[\![\texttt{repeat}\,\{\mathit{Seq}\}\,\texttt{until}\,(\psi)]\!]\,(f) \;=\; \frac{1 + \mathsf{ert}\,[\![\mathit{Seq}]\!]\,([\psi]\cdot f)}{\mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\psi])}.$$

Proof. Let $f \in \mathbb{E}$. Moreover, recall that $\texttt{repeat}\,\{\mathit{Seq}\}\,\texttt{until}\,(\psi)$ is equivalent to the program Seq; $\texttt{while}\,(\neg\psi)\,\{\mathit{Seq}\} \in \mathsf{BNL}$. Applying the semantics of ert (Table 2), we proceed as follows:

$$\mathsf{ert}\,[\![\texttt{repeat}\,\{\mathit{Seq}\}\,\texttt{until}\,(\psi)]\!]\,(f) \;=\; \mathsf{ert}\,[\![\mathit{Seq}]\!]\,(\mathsf{ert}\,[\![\texttt{while}\,(\neg\psi)\,\{\mathit{Seq}\}]\!]\,(f))$$

Since the loop body Seq is loop–free, it terminates certainly, i.e. $\mathsf{wp}\,[\![\mathit{Seq}]\!]\,(1) = 1$ (Premise 2. of Theorem 4). Together with Lemma 3.1. and 3., all premises of Theorem 4 are satisfied. Hence, we obtain a closed form for $\mathsf{ert}\,[\![\texttt{while}\,(\neg\psi)\,\{\mathit{Seq}\}]\!]\,(f)$:

$$= \;\mathsf{ert}[\![\mathit{Seq}]\!]\Bigg(\underbrace{1 + \frac{[\neg\psi]\cdot(1 + \mathsf{ert}\,[\![\mathit{Seq}]\!]\,([\psi]\cdot f))}{1 - \mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\neg\psi])} + [\psi]\cdot f}_{=:g}\Bigg)$$

By Theorem 2, we know $\mathsf{ert}\,[\![\mathit{Seq}]\!]\,(g) = \mathsf{ert}\,[\![\mathit{Seq}]\!]\,(0) + \mathsf{wp}\,[\![C]\!]\,(g)$ for any g. Thus:

$$= \;\mathsf{ert}\,[\![\mathit{Seq}]\!]\,(0) + \mathsf{wp}[\![\mathit{Seq}]\!]\Bigg(\underbrace{1 + \frac{[\neg\psi]\cdot(1 + \mathsf{ert}\,[\![\mathit{Seq}]\!]\,([\psi]\cdot f))}{1 - \mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\neg\psi])} + [\psi]\cdot f}_{g}\Bigg)$$

Since wp is linear (Theorem 1 (2)), we obtain:

$$\begin{aligned} = \;& \mathsf{ert}\,[\![\mathit{Seq}]\!]\,(0) + \underbrace{\mathsf{wp}\,[\![\mathit{Seq}]\!]\,(1)}_{=\,1} + \mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\psi]\cdot f) \\ & + \mathsf{wp}\,[\![\mathit{Seq}]\!]\left(\frac{[\neg\psi]\cdot(1 + \mathsf{ert}\,[\![\mathit{Seq}]\!]\,([\psi]\cdot f))}{1 - \mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\neg\psi])}\right) \end{aligned}$$

By a few simple algebraic transformations, this coincides with:

$$= \;1 + \mathsf{ert}\,[\![\mathit{Seq}]\!]\,(0) + \mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\psi]\cdot f) + \mathsf{wp}\,[\![\mathit{Seq}]\!]\left([\neg\psi]\cdot\frac{1 + \mathsf{ert}\,[\![\mathit{Seq}]\!]\,([\psi]\cdot f)}{1 - \mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\neg\psi])}\right)$$

Let R denote the fraction above. Then Lemma 3.1. and 2. implies $R \not\pitchfork \mathit{Seq}$. We may thus apply Lemma 1 to derive $\mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\neg\psi]\cdot R) = \mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\neg\psi])\cdot R$. Hence:

$$= \;1 + \mathsf{ert}\,[\![\mathit{Seq}]\!]\,(0) + \mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\psi]\cdot f) + \mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\neg\psi])\cdot\frac{1 + \mathsf{ert}\,[\![\mathit{Seq}]\!]\,([\psi]\cdot f)}{1 - \mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\neg\psi])}$$

Again, by Theorem 2, we know that $\mathsf{ert}\,[\![\mathit{Seq}]\!]\,(g) = \mathsf{ert}\,[\![\mathit{Seq}]\!]\,(0) + \mathsf{wp}\,[\![\mathit{Seq}]\!]\,(g)$ for any g. Thus, for $g = [\psi]\cdot f$, this yields:

$$= \;1 + \mathsf{ert}\,[\![\mathit{Seq}]\!]\,([\psi]\cdot f) + \mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\neg\psi])\cdot\frac{1 + \mathsf{ert}\,[\![\mathit{Seq}]\!]\,([\psi]\cdot f)}{1 - \mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\neg\psi])}$$

Then a few algebraic transformations lead us to the claimed ERT:

$$= \frac{1 + \mathsf{ert}\,[\![Seq]\!]\,([\psi] \cdot f)}{\mathsf{wp}\,[\![Seq]\!]\,([\psi])}. \qquad \square$$

Note that Theorem 5 holds for arbitrary postexpectations $f \in \mathbb{E}$. This enables *compositional reasoning* about ERTs of BNL programs. Since all other rules of the ert–calculus for loop–free programs amount to simple syntactical transformations (see Table 2), we conclude that

Corollary 1. *For any* $C \in \mathsf{BNL}$*, a closed form for* $\mathsf{ert}\,[\![C]\!]\,(0)$ *can be computed compositionally.*

Example 5. Theorem 5 allows us to comfortably compute the ERT of the BNL program C_{dice} introduced in Example 4:

$$x_1 :\approx \texttt{Unif}[1\ldots6];\ \texttt{repeat}\,\{x_2 :\approx \texttt{Unif}[1\ldots6]\}\,\texttt{until}\,(x_2 \geq x_1)$$

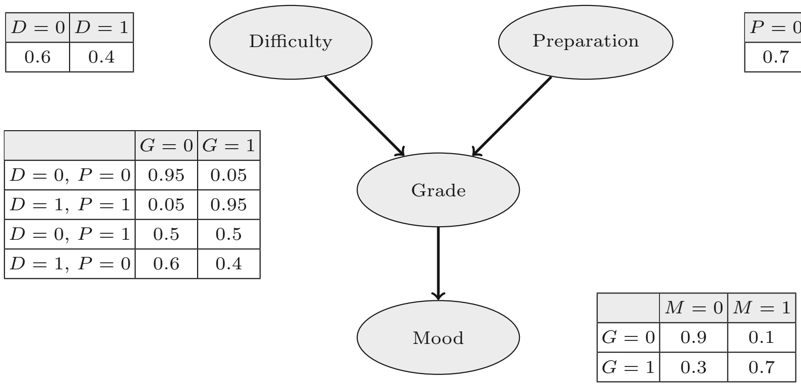
For the ERT, we have

$$\begin{aligned}
&\mathsf{ert}\,[\![C_{dice}]\!]\,(0)\\
=\ &\mathsf{ert}\,[\![x_1 :\approx \texttt{Unif}[1\ldots6]]\!]\,(\mathsf{ert}\,[\![\texttt{repeat}\,\{\ldots\}\,\texttt{until}\,([x_2 \geq x_1])]\!]\,(0)) && \text{(Table 2)}\\
=\ &\mathsf{ert}\,[\![x_1 :\approx \texttt{Unif}[1\ldots6]]\!]\left(\frac{1 + \mathsf{ert}\,[\![x_2 :\approx \texttt{Unif}[1\ldots6]]\!]\,([x_2 \geq x_1])}{\mathsf{wp}\,[\![x_1 :\approx \texttt{Unif}[1\ldots6]]\!]\,([x_2 \geq x_1])}\right) && \text{(Thm. 5)}\\
=\ &\sum_{1\leq i\leq 6} 1/6 \cdot \frac{1 + \sum_{1\leq j\leq 6} 1/6 \cdot [j \geq i]}{\sum_{1\leq j\leq 6} 1/6 \cdot [j \geq i]} && \text{(Table 2)}\\
=\ &3.45. && \triangle
\end{aligned}$$

5.2 Bayesian Networks

To reason about expected sampling times of BNs, it remains to develop a sound translation from BNs with observations into equivalent BNL programs. A BN is a probabilistic graphical model that is given by a directed acyclic graph. Every node is a random variable and a directed edge between two nodes expresses a probabilistic dependency between these nodes.

As a running example, consider the BN depicted in Fig. 3 (inspired by [31]) that models the mood of students after taking an exam. The network contains four random variables. They represent the difficulty of the exam (D), the level of preparation of a student (P), the achieved grade (G), and the resulting mood (M). For simplicity, let us assume that each random variable assumes either 0 or 1. The edges express that the student's mood depends on the achieved grade which, in turn, depends on the difficulty of the exam and the preparation of the student. Every node is accompanied by a table that provides the conditional probabilities of a node *given* the values of all the nodes it depends upon. We can then use the BN to answer queries such as "What is the probability that a



$D=0$	$D=1$
0.6	0.4

$P=0$	$P=1$
0.7	0.3

	$G=0$	$G=1$
$D=0,\ P=0$	0.95	0.05
$D=1,\ P=1$	0.05	0.95
$D=0,\ P=1$	0.5	0.5
$D=1,\ P=0$	0.6	0.4

	$M=0$	$M=1$
$G=0$	0.9	0.1
$G=1$	0.3	0.7

Fig. 3. A Bayesian network

student is well–prepared for an exam ($P = 1$), but ends up with a bad mood ($M = 0$)?"

In order to translate BNs into equivalent BNL programs, we need a formal representation first. Technically, we consider *extended* BNs in which nodes may additionally depend on inputs that are not represented by nodes in the network. This allows us to define a compositional translation without modifying conditional probability tables.

Towards a formal definition of extended BNs, we use the following notation. A tuple $(s_1, \ldots, s_k) \in S^k$ of length k over some set S is denoted by $\mathbf{s}$. The empty tuple is ε. Moreover, for $1 \leq i \leq k$, the i-th element of tuple $\mathbf{s}$ is given by $\mathbf{s}(i)$. To simplify the presentation, we assume that all nodes and all inputs are represented by natural numbers.

Definition 7. *An* extended Bayesian network*, EBN for short, is a tuple* $\mathcal{B} = (V, I, E, \mathsf{Vals}, \mathsf{dep}, \mathsf{cpt})$*, where*

- $V \subseteq \mathbb{N}$ *and* $I \subseteq \mathbb{N}$ *are finite disjoint sets of* nodes *and* inputs*.*
- $E \subseteq V \times V$ *is a set of* edges *such that* (V, E) *is a directed acyclic graph.*
- Vals *is a finite set of possible* values *that can be assigned to each node.*
- $\mathsf{dep}\colon V \to (V \cup I)^*$ *is a function assigning each node* v *to an ordered sequence of* dependencies*. That is,* $\mathsf{dep}(v) = (u_1, \ldots, u_m)$ *such that* $u_i < u_{i+1}$ $(1 \leq i < m)$*. Moreover, every dependency* u_j $(1 \leq j \leq m)$ *is either an input, i.e.* $u_j \in I$*, or a node with an edge to* v*, i.e.* $u_j \in V$ *and* $(u_j, v) \in E$*.*
- cpt *is a function mapping each node* v *to its* conditional probability table $\mathsf{cpt}[v]$*. That is, for* $k = |\mathsf{dep}(v)|$*,* $\mathsf{cpt}[v]$ *is given by a function of the form*

$$\mathsf{cpt}[v] : \mathsf{Vals}^k \to \mathsf{Vals} \to [0,1] \quad \textit{such that} \quad \sum_{\mathbf{z} \in \mathsf{Vals}^k, a \in \mathsf{Vals}} \mathsf{cpt}[v](\mathbf{z})(a) \;=\; 1.$$

Here, the i-th entry in a tuple $\mathbf{z} \in \mathsf{Vals}^k$ *corresponds to the value assigned to the i-th entry in the sequence of dependencies* $\mathsf{dep}(v)$*.*

A Bayesian network *(BN) is an extended BN without inputs, i.e. $I = \emptyset$. In particular, the dependency function is of the form* $\mathsf{dep}\colon V \to V^*$.

Example 6. The formalization of our example BN (Fig. 3) is straightforward. For instance, the dependencies of variable G are given by $\mathsf{dep}(G) = (D, P)$ (assuming D is encoded by an integer less than P). Furthermore, every entry in the conditional probability table of node G corresponds to an evaluation of the function $\mathsf{cpt}[G]$. For example, if $D = 1$, $P = 0$, and $G = 1$, we have $\mathsf{cpt}[G](1, 0)(1) = 0.4$. △

In general, the conditional probability table cpt determines the conditional probability distribution of each node $v \in V$ given the nodes and inputs it depends on. Formally, we interpret an entry in a conditional probability table as follows:

$$\mathsf{Pr}\,(v = a \,|\, \mathsf{dep}(v) = \mathbf{z}) \;=\; \mathsf{cpt}[v](\mathbf{z})(a),$$

where $v \in V$ is a node, $a \in \mathsf{Vals}$ is a value, and $\mathbf{z}$ is a tuple of values of length $|\mathsf{dep}(v)|$. Then, by the chain rule, the joint probability of a BN is given by the product of its conditional probability tables (cf. [4]).

Definition 8. *Let $\mathcal{B} = (V, I, E, \mathit{Vals}, \mathsf{dep}, \mathsf{cpt})$ be an extended Bayesian network. Moreover, let $W \subseteq V$ be a downward closed*[5] *set of nodes. With each $w \in W \cup I$, we associate a fixed value $\underline{w} \in \mathit{Vals}$. This notation is lifted pointwise to tuples of nodes and inputs. Then the* joint probability *in which nodes in W assume values $\underline{W}$ is given by*

$$\mathsf{Pr}\,(W = \underline{W}) \;=\; \prod_{v \in W} \mathsf{Pr}\left(v = \underline{v} \,|\, \mathsf{dep}(v) = \underline{\mathsf{dep}(v)}\right) \;=\; \prod_{v \in W} \mathsf{cpt}[v](\underline{\mathsf{dep}(v)})(\underline{v}).$$

The conditional joint probability distribution of a set of nodes W, given observations on a set of nodes O, is then given by the quotient $\mathsf{Pr}(W=\underline{W})/\mathsf{Pr}(O=\underline{O})$.

For example, the probability of a student having a bad mood, i.e. $M = 0$, after getting a bad grade ($G = 0$) for an easy exam ($D = 0$) given that she was well–prepared, i.e. $P = 1$, is

$$\begin{aligned}\mathsf{Pr}\,(D = 0, G = 0, M = 0 \mid P = 1) \;&=\; \frac{\mathsf{Pr}\,(D = 0, G = 0, M = 0, P = 1)}{\mathsf{Pr}\,(P = 1)} \\ &=\; \frac{0.9 \cdot 0.5 \cdot 0.6 \cdot 0.3}{0.3} \;=\; 0.27.\end{aligned}$$

5.3 From Bayesian Networks to BNL

We now develop a compositional translation from EBNs into BNL programs. Throughout this section, let $\mathcal{B} = (V, I, E, \mathsf{Vals}, \mathsf{dep}, \mathsf{cpt})$ be a fixed EBN. Moreover, with every node or input $v \in V \cup I$ we associate a program variable x_v.

We proceed in three steps: First, *every node together with its dependencies* is translated into a *block* of a BNL program. These blocks are then composed into a single BNL program that captures the whole BN. Finally, we implement conditioning by means of rejection sampling.

[5] W is downward closed if $v \in W$ and $(u, v) \in E$ implies $u \in E$.

Step 1: We first present the atomic building blocks of our translation. Let $v \in V$ be a node. Moreover, let $\mathbf{z} \in \mathsf{Vals}^{|\mathsf{dep}(v)|}$ be an evaluation of the dependencies of v. That is, $\mathbf{z}$ is a tuple that associates a value with every node and input that v depends on (in the same order as $\mathsf{dep}(v)$). For every node v and evaluation of its dependencies $\mathbf{z}$, we define a corresponding guard and a random assignment:

$$\begin{aligned} \mathit{guard}_{\mathcal{B}}(v, \mathbf{z}) &= \bigwedge_{1 \leq i \leq |\mathsf{dep}(v)|} x_{\mathsf{dep}(v)(i)} = \mathbf{z}(i) \\ \mathit{assign}_{\mathcal{B}}(v, \mathbf{z}) &= x_v :\approx \sum_{a \in \mathsf{Vals}} \mathsf{cpt}[v](\mathbf{z})(a) \cdot \langle a \rangle \end{aligned}$$

Note that $\mathsf{dep}(v)(i)$ is the i-th element from the sequence of nodes $\mathsf{dep}(v)$.

Example 7. Continuing our previous example (see Fig. 1), assume we fixed the node $v = G$. Moreover, let $\mathbf{z} = (1, 0)$ be an evaluation of $\mathsf{dep}(v) = (S, R)$. Then the guard and assignment corresponding to v and $\mathbf{z}$ are given by:

$$\begin{aligned} \mathit{guard}_{\mathcal{B}}(G, (1,0)) &= x_D = 1 \ \wedge x_P = 0, \text{ and} \\ \mathit{assign}_{\mathcal{B}}(G, (1,0)) &= x_G :\approx 0.6 \cdot \langle 0 \rangle + 0.4 \cdot \langle 1 \rangle. \end{aligned} \qquad \triangle$$

We then translate every node $v \in V$ into a program block that uses guards to determine the rows in the conditional probability table under consideration. After that, the program samples from the resulting probability distribution using the previously constructed assignments. In case a node does neither depend on other nodes nor input variables we omit the guards. Formally,

$$\mathit{block}_{\mathcal{B}}(v) = \begin{cases} \mathit{assign}_{\mathcal{B}}(v, \varepsilon) & \text{if } |\mathsf{dep}(v)| = 0 \\ \texttt{if}\, (\mathit{guard}_{\mathcal{B}}(v, \mathbf{z_1}))\, \{ & \\ \quad \mathit{assign}_{\mathcal{B}}(v, \mathbf{z_1})\} & \\ \texttt{else}\, \{\texttt{if}\, (\mathit{guard}_{\mathcal{B}}(v, \mathbf{z_2}))\, \{ & \text{if } |\mathsf{dep}(v)| = k > 0 \\ \quad \mathit{assign}_{\mathcal{B}}(v, \mathbf{z_2})\} & \text{and } \mathsf{Vals}^k = \{\mathbf{z_1}, \ldots, \mathbf{z_m}\}. \\ \ldots\} \;\texttt{else}\; \{ & \\ \quad \mathit{assign}_{\mathcal{B}}(v, \mathbf{z_m})\} \ldots\} & \end{cases}$$

Remark 1. The guards under consideration are conjunctions of equalities between variables and literals. We could thus use a more efficient translation of conditional probability tables by adding a `switch-case` statement to our probabilistic programming language. Such a statement is of the form

$$\texttt{switch}(\mathbf{x})\ \{\ \texttt{case}\ \mathbf{a}_1 : C_1\ \texttt{case}\ a_2 : C_2\ \ldots\ \texttt{default} : C_m\},$$

where $\mathbf{x}$ is a tuple of variables, and $\mathbf{a}_1, \ldots \mathbf{a}_{m-1}$ are tuples of rational numbers of the same length as $\mathbf{x}$. With respect to the `wp` semantics, a `switch-case` statement is syntactic sugar for nested `if-then-else` blocks as used in the above translation. However, the runtime model of a `switch-case` statement requires just a single guard evaluation (φ) instead of potentially multiple guard evaluations when evaluating nested `if-then-else` blocks. Since the above adaption is straightforward, we opted to use nested `if-then-else` blocks to keep our programming language simple and allow, in principle, more general guards. $\triangle$

Step 2: The next step is to translate a complete EBN into a BNL program. To this end, we compose the blocks obtained from each node starting at the roots of the network. That is, all nodes that contain no incoming edges. Formally,

$$\mathit{roots}(\mathcal{B}) = \{v \in V_{\mathcal{B}} \mid \neg\exists u \in V_{\mathcal{B}} \colon (u,v) \in E_{\mathcal{B}}\}.$$

After translating every node in the network, we remove them from the graph, i.e. every root becomes an input, and proceed with the translation until all nodes have been removed. More precisely, given a set of nodes $S \subseteq V$, the extended BN $\mathcal{B} \setminus S$ obtained by removing S from $\mathcal{B}$ is defined as

$$\mathcal{B} \setminus S \;=\; (V \setminus S,\, I \cup S,\, E \setminus (V \times S \cup S \times V),\, \mathsf{dep},\, \mathsf{cpt})\,.$$

With these auxiliary definitions readily available, an extended BN $\mathcal{B}$ is translated into a BNL program as follows:

$$\mathit{BNL}(\mathcal{B}) \;=\; \begin{cases} \mathit{block}_{\mathcal{B}}(r_1); \ldots ; \mathit{block}_{\mathcal{B}}(r_m) & \text{if } \mathit{roots}(\mathcal{B}) = \{r_1, \ldots, r_m\} = V \\ \mathit{block}_{\mathcal{B}}(r_1); \ldots ; \mathit{block}_{\mathcal{B}}(r_m); & \text{if } \mathit{roots}(\mathcal{B}) = \{r_1, \ldots, r_m\} \subsetneq V \\ \mathit{BNL}(\mathcal{B} \setminus \mathit{roots}(\mathcal{B})) & \end{cases}$$

Step 3: To complete the translation, it remains to account for observations. Let $\mathit{cond} : V \to \mathsf{Vals} \cup \{\bot\}$ be a function mapping every node either to an observed value in Vals or to $\bot$. The former case is interpreted as an observation that node v has value $\mathit{cond}(v)$. Otherwise, i.e. if $\mathit{cond}(v) = \bot$, the value of node v is *not observed*. We collect all observed nodes in the set $O = \{v \in V \mid \mathit{cond}(v) \neq \bot\}$. It is then natural to incorporate conditioning into our translation by applying rejection sampling: We repeatedly execute a BNL program until every observed node has the desired value $\mathit{cond}(v)$. In the presence of observations, we translate the extended BN $\mathcal{B}$ into a BNL program as follows:

$$\mathit{BNL}(\mathcal{B}, \mathit{cond}) \;=\; \texttt{repeat}\,\{\mathit{BNL}(\mathcal{B})\}\,\texttt{until}\left(\bigwedge_{v \in O} x_v = \mathit{cond}(v)\right)$$

Example 8. Consider, again, the BN $\mathcal{B}$ depicted in Fig. 3. Moreover, assume we observe $P = 1$. Hence, the conditioning function cond is given by $\mathit{cond}(P) = 1$ and $\mathit{cond}(v) = \bot$ for $v \in \{D, G, M\}$. Then the translation of $\mathcal{B}$ and cond, i.e. $\mathit{BNL}(\mathcal{B}, \mathit{cond})$, is the BNL program C_{mood} depicted in Fig. 4. △

Since our translation yields a BNL program for any given BN, we can compositionally compute a closed form for the expected simulation time of a BN. This is an immediate consequence of Corollary 1.

We still have to prove, however, that our translation is sound, i.e. the conditional joint probabilities inferred from a BN coincide with the (conditional) joint probabilities from the corresponding BNL program. Formally, we obtain the following soundness result.

```
repeat {
    $x_D :\approx 0.6 \cdot \langle 0 \rangle + 0.4 \cdot \langle 1 \rangle$;
    $x_P :\approx 0.7 \cdot \langle 0 \rangle + 0.3 \cdot \langle 1 \rangle$
    if ($x_D = 0 \wedge x_P = 0$) {
        $x_G :\approx 0.95 \cdot \langle 0 \rangle + 0.05 \cdot \langle 1 \rangle$
    } else if ($x_D = 1 \wedge x_P = 1$) {
        $x_G :\approx 0.05 \cdot \langle 0 \rangle + 0.95 \cdot \langle 1 \rangle$
    } else if ($x_D = 0 \wedge x_P = 1$) {
        $x_G :\approx 0.5 \cdot \langle 0 \rangle + 0.5 \cdot \langle 1 \rangle$
    } else {
        $x_G :\approx 0.6 \cdot \langle 0 \rangle + 0.4 \cdot \langle 1 \rangle$
    };
    if ($x_G = 0$) {
        $x_M :\approx 0.9 \cdot \langle 0 \rangle + 0.1 \cdot \langle 1 \rangle$
    } else {
        $x_M :\approx 0.3 \cdot \langle 0 \rangle + 0.7 \cdot \langle 1 \rangle$
    }
} until ($x_P = 1$)
```

Fig. 4. The BNL program C_{mood} obtained from the BN in Fig. 3.

Theorem 6 (Soundness of Translation). *Let $\mathcal{B} = (V, I, E, \mathsf{Vals}, \mathsf{dep}, \mathsf{cpt})$ be a BN and cond : $V \to \mathsf{Vals} \cup \{\bot\}$ be a function determining the observed nodes. For each node and input v, let $\underline{v} \in \mathsf{Vals}$ be a fixed value associated with v. In particular, we set $\underline{v} = cond(v)$ for each observed node $v \in O$. Then*

$$\mathsf{wp}\,[\![BNL(\mathcal{B}, cond)]\!]\left(\left[\bigwedge_{v \in V \setminus O} x_v = \underline{v}\right]\right) = \frac{\mathsf{Pr}\left(\bigwedge_{v \in V} v = \underline{v}\right)}{\mathsf{Pr}\left(\bigwedge_{o \in O} o = \underline{o}\right)}.$$

Proof. Without conditioning, i.e. $O = \emptyset$, the proof proceeds by induction on the number of nodes of $\mathcal{B}$. With conditioning, we additionally apply Theorems 3 and 5 to deal with loops introduced by observed nodes. See [3, Appendix A.7]. □

Example 9 (Expected Sampling Time of a BN). Consider, again, the BN $\mathcal{B}$ in Fig. 3. Moreover, recall the corresponding program C_{mood} derived from $\mathcal{B}$ in Fig. 4, where we observed $P = 1$. By Theorem 6 we can also determine the probability that a student who got a bad grade in an easy exam was well–prepared by means of weakest precondition reasoning. This yields

$$\begin{aligned}&\mathsf{wp}\,[\![C_{mood}]\!]\,([x_D = 0 \wedge x_G = 0 \wedge x_M = 0])\\ &= \frac{\mathsf{Pr}\,(D = 0, G = 0, M = 0, P = 1)}{\mathsf{Pr}\,(P = 1)} = 0.27.\end{aligned}$$

Furthermore, by Corollary 1, it is straightforward to determine the expected time to obtain a single sample of $\mathcal{B}$ that satisfies the observation $P = 1$:

$$\mathsf{ert}\,[\![C_{mood}]\!]\,(0) = \frac{1 + \mathsf{ert}\,[\![C_{loop\text{-}body}]\!]\,(0)}{\mathsf{wp}\,[\![C_{loop\text{-}body}]\!]\,([P = 1])} = 23.4 + {}^{1}\!/_{15} = 23.4\bar{6}. \qquad \triangle$$

6 Implementation

We implemented a prototype in Java to analyze expected sampling times of Bayesian networks. More concretely, our tool takes as input a BN together with observations in the popular Bayesian Network Interchange Format.[6] The BN is then translated into a BNL program as shown in Sect. 5. Our tool applies the ert–calculus together with our proof rules developed in Sect. 4 to compute the exact expected runtime of the BNL program.

The size of the resulting BNL program is linear in the total number of rows of all conditional probability tables in the BN. The program size is thus *not* the bottleneck of our analysis. As we are dealing with an NP–hard problem [12,13], it is not surprising that our algorithm has a worst–case exponential time complexity. However, also the space complexity of our algorithm is exponential in the worst case: As an expectation is propagated backwards through an `if`–clause of the BNL program, the size of the expectation is potentially multiplied. This is also the reason that our analysis runs out of memory on some benchmarks.

We evaluated our implementation on the *largest* BNs in the Bayesian Network Repository [46] that consists—to a large extent—of real–world BNs including expert systems for, e.g., electromyography (`munin`) [2], hematopathology diagnosis (`hepar2`) [42], weather forecasting (`hailfinder`) [1], and printer troubleshooting in Windows 95 (`win95pts`) [45, Sect. 5.6.2]. For a evaluation of *all* BNs in the repository, we refer to the extended version of this paper [3, Sect. 6].

All experiments were performed on an HP BL685C G7. Although up to 48 cores with 2.0 GHz were available, only one core was used apart from Java's garbage collection. The Java virtual machine was limited to 8 GB of RAM.

Our experimental results are shown in Table 3. The number of nodes of the considered BNs ranges from 56 to 1041. For each Bayesian network, we computed the expected sampling time (EST) for different collections of observed nodes (#obs). Furthermore, Table 3 provides the *average Markov Blanket size*, i.e. the average number of parents, children and children's parents of nodes in the BN [43], as an indicator measuring how independent nodes in the BN are.

Observations were picked at random. Note that the time required by our prototype varies depending on both the number of observed nodes and the actual observations. Thus, there are cases in which we run out of memory although the total number of observations is small.

In order to obtain an understanding of what the EST corresponds to in actual execution times on a real machine, we also performed simulations for the `win95pts` network. More precisely, we generated Java programs from this network analogously to the translation in Sect. 5. This allowed us to approximate that our Java setup can execute $9.714 \cdot 10^6$ steps (in terms of EST) per second.

For the `win95pts` with 17 observations, an EST of $1.11 \cdot 10^{15}$ then corresponds to an expected time of approximately 3.6 *years* in order to obtain a *single* valid sample. We were additionally able to find a case with 13 observed nodes where our tool discovered within 0.32 s an EST that corresponds to approximately 4.3 *million years*. In contrast, exact inference using variable elimination was almost instantaneous. This demonstrates that knowing expected sampling times upfront can indeed be beneficial when selecting an inference method.

[6] http://www.cs.cmu.edu/~fgcozman/Research/InterchangeFormat/.

Table 3. Experimental results. Time is in seconds. MO denotes out of memory.

BN	#obs	Time	EST	#obs	Time	EST	#obs	Time	EST
`hailfinder`	*#nodes: 56, #edges: 66, avg. Markov Blanket: 3.54*								
	0	0.23	$9.500 \cdot 10^1$	5	0.63	$5.016 \cdot 10^5$	9	0.46	$9.048 \cdot 10^6$
`hepar2`	*#nodes: 70, #edges: 123, avg. Markov Blanket: 4.51*								
	0	0.22	$1.310 \cdot 10^2$	1	1.84	$1.579 \cdot 10^2$	2	MO	–
`win95pts`	*#nodes: 76, #edges: 112, avg. Markov Blanket: 5.92*								
	0	0.20	$1.180 \cdot 10^2$	1	0.36	$2.284 \cdot 10^3$	3	0.36	$4.296 \cdot 10^5$
	7	0.91	$1.876 \cdot 10^6$	12	0.42	$3.973 \cdot 10^7$	17	61.73	$1.110 \cdot 10^{15}$
`pathfinder`	*#nodes: 135, #edges: 200, avg. Markov Blanket: 3.04*								
	0	0.37	217	1	0.53	$1.050 \cdot 10^4$	3	31.31	$2.872 \cdot 10^4$
	5	MO	–	7	5.44	∞	7	480.83	∞
`andes`	*#nodes: 223, #edges: 338, avg. Markov Blanket: 5.61*								
	0	0.46	$3.570 \cdot 10^2$	1	MO	–	3	1.66	$5.251 \cdot 10^3$
	5	1.41	$9.862 \cdot 10^3$	7	0.99	$8.904 \cdot 10^4$	9	0.90	$6.637 \cdot 10^5$
`pigs`	*#nodes: 441, #edges: 592, avg. Markov Blanket: 3.66*								
	0	0.57	$7.370 \cdot 10^2$	1	0.74	$2.952 \cdot 10^3$	3	0.88	$2.362 \cdot 10^3$
	5	0.85	$1.260 \cdot 10^5$	7	1.02	$1.511 \cdot 10^6$	8	MO	–
`munin`	*#nodes: 1041, #edges: 1397, avg. Markov Blanket: 3.54*								
	0	1.29	$1.823 \cdot 10^3$	1	1.47	$3.648 \cdot 10^4$	3	1.37	$1.824 \cdot 10^7$
	5	1.43	∞	9	1.79	$1.824 \cdot 10^{16}$	10	65.64	$1.153 \cdot 10^{18}$

7 Conclusion

We presented a syntactic notion of independent and identically distributed probabilistic loops and derived dedicated proof rules to determine exact expected outcomes and runtimes of such loops. These rules do not require any user–supplied information, such as invariants, (super)martingales, etc.

Moreover, we isolated a syntactic fragment of probabilistic programs that allows to compute expected runtimes in a highly automatable fashion. This fragment is non–trivial: We show that all Bayesian networks can be translated into programs within this fragment. Hence, we obtain an automated formal method for computing expected simulation times of Bayesian networks. We implemented this method and successfully applied it to various real–world BNs that stem from, amongst others, medical applications. Remarkably, our tool was capable of proving extremely large expected sampling times within seconds.

There are several directions for future work: For example, there exist subclasses of BNs for which exact inference is in P, e.g. polytrees. Are there analogies for probabilistic programs? Moreover, it would be interesting to consider more complex graphical models, such as recursive BNs [16].

References

1. Abramson, B., Brown, J., Edwards, W., Murphy, A., Winkler, R.L.: Hailfinder: a Bayesian system for forecasting severe weather. Int. J. Forecast. **12**(1), 57–71 (1996)
2. Andreassen, S., Jensen, F.V., Andersen, S.K., Falck, B., Kjærulff, U., Woldbye, M., Sørensen, A., Rosenfalck, A., Jensen, F.: MUNIN: an expert EMG Assistant. In: Computer-Aided Electromyography and Expert Systems, pp. 255–277. Pergamon Press (1989)
3. Batz, K., Kaminski, B.L., Katoen, J., Matheja, C.: How long, O Bayesian network, will I sample thee? arXiv extended version (2018)
4. Bishop, C.: Pattern Recognition and Machine Learning. Springer, New York (2006)
5. 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
6. Brázdil, T., Kiefer, S., Kucera, A., Vareková, I.H.: Runtime analysis of probabilistic programs with unbounded recursion. J. Comput. Syst. Sci. **81**(1), 288–310 (2015)
7. Celiku, O., McIver, A.: Compositional specification and analysis of cost-based properties in probabilistic programs. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 107–122. Springer, Heidelberg (2005). https://doi.org/10.1007/11526841_9
8. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 511–526. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_34
9. Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In: POPL, pp. 327–342. ACM (2016)
10. Chatterjee, K., Novotný, P., Zikelic, D.: Stochastic invariants for probabilistic termination. In: POPL, pp. 145–160. ACM (2017)
11. Constantinou, A.C., Fenton, N.E., Neil, M.: pi-football: a Bayesian network model for forecasting association football match outcomes. Knowl. Based Syst. **36**, 322–339 (2012)
12. Cooper, G.F.: The computational complexity of probabilistic inference using Bayesian belief networks. Artif. Intell. **42**(2–3), 393–405 (1990)
13. Dagum, P., Luby, M.: Approximating probabilistic inference in Bayesian belief networks is NP-hard. Artif. Intell. **60**(1), 141–153 (1993)
14. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
15. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Upper Saddle River (1976)
16. Etessami, K., Yannakakis, M.: Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. JACM **56**(1), 1:1–1:66 (2009)
17. Fioriti, L.M.F., Hermanns, H.: Probabilistic termination: soundness, completeness, and compositionality. In: POPL, pp. 489–501. ACM (2015)
18. Friedman, N., Linial, M., Nachman, I., Pe'er, D.: Using Bayesian networks to analyze expression data. In: RECOMB, pp. 127–135. ACM (2000)
19. Frohn, F., Naaf, M., Hensel, J., Brockschmidt, M., Giesl, J.: Lower runtime bounds for integer programs. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 550–567. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_37

20. Goodman, N.D.: The principles and practice of probabilistic programming. In: POPL, pp. 399–402. ACM (2013)
21. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: A language for generative models. In: UAI, pp. 220–229. AUAI Press (2008)
22. Gordon, A.D., Graepel, T., Rolland, N., Russo, C.V., Borgström, J., Guiver, J.: Tabular: a schema-driven probabilistic programming language. In: POPL, pp. 321–334. ACM (2014)
23. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Future of Software Engineering, pp. 167–181. ACM (2014)
24. Heckerman, D.: A tutorial on learning with Bayesian networks. In: Holmes, D.E., Jain, L.C. (eds.) Innovations in Bayesian Networks. Studies in Computational Intelligence, vol. 156, pp. 33–82. Springer, Heidelberg (2008)
25. Hehner, E.C.R.: A probability perspective. Formal Aspects Comput. **23**(4), 391–419 (2011)
26. Hoffman, M.D., Gelman, A.: The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. J. Mach. Learn. Res. **15**(1), 1593–1623 (2014)
27. Jiang, X., Cooper, G.F.: A Bayesian spatio-temporal method for disease outbreak detection. JAMIA **17**(4), 462–471 (2010)
28. Kaminski, B.L., Katoen, J.-P.: On the hardness of almost–sure termination. In: Italiano, G.F., Pighizzini, G., Sannella, D.T. (eds.) MFCS 2015. LNCS, vol. 9234, pp. 307–318. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48057-1_24
29. Kaminski, B.L., Katoen, J.: A weakest pre-expectation semantics for mixed-sign expectations. In: LICS (2017)
30. Kaminski, B.L., Katoen, J.-P., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run–times of probabilistic programs. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 364–389. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_15
31. Koller, D., Friedman, N.: Probabilistic Graphical Models - Principles and Techniques. MIT Press, Cambridge (2009)
32. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. **22**(3), 328–350 (1981)
33. Kozen, D.: A probabilistic PDL. J. Comput. Syst. Sci. **30**(2), 162–178 (1985)
34. Lassez, J.L., Nguyen, V.L., Sonenberg, L.: Fixed point theorems and semantics: a folk tale. Inf. Process. Lett. **14**(3), 112–116 (1982)
35. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Springer, New York (2004). http://doi.org/10.1007/b138392
36. Minka, T., Winn, J.: Infer.NET (2017). http://infernet.azurewebsites.net/. Accessed Oct 17
37. Minka, T., Winn, J.M.: Gates. In: NIPS, pp. 1073–1080. Curran Associates (2008)
38. Monniaux, D.: An abstract analysis of the probabilistic termination of programs. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 111–126. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-47764-0_7
39. Neapolitan, R.E., Jiang, X.: Probabilistic Methods for Financial and Marketing Informatics. Morgan Kaufmann, Burlington (2010)
40. Nori, A.V., Hur, C., Rajamani, S.K., Samuel, S.: R2: an efficient MCMC sampler for probabilistic programs. In: AAAI, pp. 2476–2482. AAAI Press (2014)
41. Olmedo, F., Kaminski, B.L., Katoen, J., Matheja, C.: Reasoning about recursive probabilistic programs. In: LICS, pp. 672–681. ACM (2016)

42. Onisko, A., Druzdzel, M.J., Wasyluk, H.: A probabilistic causal model for diagnosis of liver disorders. In: Proceedings of the Seventh International Symposium on Intelligent Information Systems (IIS-98), pp. 379–387 (1998)
43. Pearl, J.: Bayesian networks: a model of self-activated memory for evidential reasoning. In: Proceedings of CogSci, pp. 329–334 (1985)
44. Pfeffer, A.: Figaro: an object-oriented probabilistic programming language. Charles River Analytics Technical Report 137, 96 (2009)
45. Ramanna, S., Jain, L.C., Howlett, R.J.: Emerging Paradigms in Machine Learning. Springer, Heidelberg (2013)
46. Scutari, M.: Bayesian Network Repository (2017). http://www.bnlearn.com
47. Sharir, M., Pnueli, A., Hart, S.: Verification of probabilistic programs. SIAM J. Comput. **13**(2), 292–314 (1984)
48. Wood, F., van de Meent, J., Mansinghka, V.: A new approach to probabilistic programming inference. In: JMLR Workshop and Conference Proceedings, AISTATS, vol. 33, pp. 1024–1032 (2014). JMLR.org
49. Yuan, C., Druzdzel, M.J.: Importance sampling algorithms for Bayesian networks: principles and performance. Math. Comput. Model. **43**(9–10), 1189–1207 (2006)
50. Zweig, G., Russell, S.J.: Speech recognition with dynamic Bayesian networks. In: AAAI/IAAI, pp. 173–180. AAAI Press/The MIT Press (1998)

Relational Reasoning for Markov Chains in a Probabilistic Guarded Lambda Calculus

Alejandro Aguirre[1(✉)], Gilles Barthe[1], Lars Birkedal[2], Aleš Bizjak[2], Marco Gaboardi[3], and Deepak Garg[4]

[1] IMDEA Software Institute, Madrid, Spain
alejandro.aguirre@imdea.org
[2] Aarhus University, Aarhus, Denmark
[3] University at Buffalo, SUNY, Buffalo, USA
[4] MPI-SWS, Kaiserslautern and Saarbrücken, Germany

Abstract. We extend the simply-typed guarded λ-calculus with discrete probabilities and endow it with a program logic for reasoning about relational properties of guarded probabilistic computations. This provides a framework for programming and reasoning about infinite stochastic processes like Markov chains. We demonstrate the logic sound by interpreting its judgements in the topos of trees and by using probabilistic couplings for the semantics of relational assertions over distributions on discrete types.

The program logic is designed to support syntax-directed proofs in the style of relational refinement types, but retains the expressiveness of higher-order logic extended with discrete distributions, and the ability to reason relationally about expressions that have different types or syntactic structure. In addition, our proof system leverages a well-known theorem from the coupling literature to justify better proof rules for relational reasoning about probabilistic expressions. We illustrate these benefits with a broad range of examples that were beyond the scope of previous systems, including shift couplings and lump couplings between random walks.

1 Introduction

Stochastic processes are often used in mathematics, physics, biology or finance to model evolution of systems with uncertainty. In particular, Markov chains are "memoryless" stochastic processes, in the sense that the evolution of the system depends only on the current state and not on its history. Perhaps the most emblematic example of a (discrete time) Markov chain is the simple random walk over the integers, that starts at 0, and that on each step moves one position either left or right with uniform probability. Let p_i be the position at time i. Then, this Markov chain can be described as:

$$p_0 = 0 \qquad p_{i+1} = \begin{cases} p_i + 1 \text{ with probability } 1/2 \\ p_i - 1 \text{ with probability } 1/2 \end{cases}$$

The goal of this paper is to develop a programming and reasoning framework for probabilistic computations over infinite objects, such as Markov chains. Although programming and reasoning frameworks for infinite objects and probabilistic computations are well-understood in isolation, their combination is challenging. In particular, one must develop a proof system that is powerful enough for proving interesting properties of probabilistic computations over infinite objects, and practical enough to support effective verification of these properties.

Modelling Probabilistic Infinite Objects. A first challenge is to model probabilistic infinite objects. We focus on the case of Markov chains, due to its importance. A (discrete-time) Markov chain is a sequence of random variables $\{X_i\}$ over some fixed type T satisfying some independence property. Thus, the straightforward way of modelling a Markov chain is as a *stream of distributions* over T. Going back to the simple example outlined above, it is natural to think about this kind of *discrete-time* Markov chain as characterized by the sequence of positions $\{p_i\}_{i \in \mathbb{N}}$, which in turn can be described as an infinite set indexed by the natural numbers. This suggests that a natural way to model such a Markov chain is to use *streams* in which each element is produced *probabilistically* from the previous one. However, there are some downsides to this representation. First of all, it requires explicit reasoning about probabilistic dependency, since X_{i+1} depends on X_i. Also, we might be interested in global properties of the executions of the Markov chain, such as "The probability of passing through the initial state infinitely many times is 1". These properties are naturally expressed as properties of the whole stream. For these reasons, we want to represent Markov chains as *distributions over streams*. Seemingly, one downside of this representation is that the set of streams is not countable, which suggests the need for introducing heavy measure-theoretic machinery in the semantics of the programming language, even when the underlying type is discrete or finite.

Fortunately, measure-theoretic machinery can be avoided (for discrete distributions) by developing a probabilistic extension of the simply-typed guarded λ-calculus and giving a semantic interpretation in the topos of trees [1]. Informally, the simply-typed guarded λ-calculus [1] extends the simply-typed lambda calculus with a *later* modality, denoted by $\triangleright$. The type $\triangleright A$ ascribes expressions that are available one unit of logical time in the future. The $\triangleright$ modality allows one to model infinite types by using "finite" approximations. For example, a stream of natural numbers is represented by the sequence of its (increasing) prefixes in the topos of trees. The prefix containing the first i elements has the type $S_i \triangleq \mathbb{N} \times \triangleright\mathbb{N} \times \ldots \times \triangleright^{(i-1)}\mathbb{N}$, representing that the first element is available now, the second element a unit time in the future, and so on. This is the key to representing probability distributions over infinite objects without measure-theoretic semantics: We model probability distributions over non-discrete sets as discrete distributions over their (the sets') approximations. For example, a distribution over streams of natural numbers (which a priori would be non-discrete since the set of streams is uncountable) would be modelled by a *sequence of distributions* over the finite approximations $S_1, S_2, \ldots$ of streams. Importantly, since each S_i is countable, each of these distributions can be discrete.

Reasoning About Probabilistic Computations. Probabilistic computations exhibit a rich set of properties. One natural class of properties is related to probabilities of events, saying, for instance, that the probability of some event E (or of an indexed family of events) increases at every iteration. However, several interesting properties of probabilistic computation, such as stochastic dominance or convergence (defined below) are relational, in the sense that they refer to two runs of two processes. In principle, both classes of properties can be proved using a higher-order logic for probabilistic expressions, e.g. the internal logic of the topos of trees, suitably extended with an axiomatization of finite distributions. However, we contend that an alternative approach inspired from refinement types is desirable and provides better support for effective verification. More specifically, reasoning in a higher-order logic, e.g. in the internal logic of the topos of trees, does not exploit the *structure of programs* for non-relational reasoning, nor the *structural similarities* between programs for relational reasoning. As a consequence, reasoning is more involved. To address this issue, we define a relational proof system that exploits the structure of the expressions and supports syntax-directed proofs, with necessary provisions for escaping the syntax-directed discipline when the expressions do not have the same structure. The proof system manipulates judgements of the form:

$$\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash t_1 : A_1 \sim t_2 : A_2 \mid \phi$$

where Δ and Γ are two typing contexts, Σ and Ψ respectively denote sets of assertions over variables in these two contexts, t_1 and t_2 are well-typed expressions of type A_1 and A_2, and ϕ is an assertion that may contain the special variables $\mathbf{r}_1$ and $\mathbf{r}_2$ that respectively correspond to the values of t_1 and t_2. The context Δ and Γ, the terms t_1 and t_2 and the types A_1 and A_2 provide a specification, while Σ, Ψ, and ϕ are useful for reasoning about relational properties over t_1, t_2, their inputs and their outputs. This form of judgement is similar to that of Relational Higher-Order Logic [2], from which our system draws inspiration.

In more detail, our relational logic comes with typing rules that allow one to reason about relational properties by exploiting as much as possible the syntactic similarities between t_1 and t_2, and to fall back on pure logical reasoning when these are not available. In order to apply relational reasoning to guarded computations the logic provides relational rules for the later modality $\triangleright$ and for a related modality $\square$, called "constant". These rules allow the relational verification of general relational properties that go beyond the traditional notion of program equivalence and, moreover, they allow the verification of properties of guarded computations over different types. The ability to reason about computations of different types provides significant benefits over alternative formalisms for relational reasoning. For example, it enables reasoning about relations between programs working on different data structures, e.g. a relation between a program working on a stream of natural numbers, and a program working on a stream of pairs of natural numbers, or having different structures, e.g. a relation between an application and a case expression.

Importantly, our approach for reasoning formally about probabilistic computations is based on *probabilistic couplings*, a standard tool from the analysis of Markov chains [3,4]. From a verification perspective, probabilistic couplings go beyond equivalence properties of probabilistic programs, which have been studied extensively in the verification literature, and yet support compositional reasoning [5,6]. The main attractive feature of coupling-based reasoning is that it limits the need of explicitly reasoning about the probabilities—this avoids complex verification conditions. We provide sound proof rules for reasoning about probabilistic couplings. Our rules make several improvements over prior relational verification logics based on couplings. First, we support reasoning over probabilistic processes of different types. Second, we use Strassen's theorem [7] a remarkable result about probabilistic couplings, to achieve greater expressivity. Previous systems required to prove a bijection between the sampling spaces to show the existence of a coupling [5,6], Strassen's theorem gives a way to show their existence which is applicable in settings where the bijection-based approach cannot be applied. And third, we support reasoning with what are called shift couplings, coupling which permits to relate the states of two Markov chains at possibly different times (more explanations below).

Case Studies. We show the flexibility of our formalism by verifying several examples of relational properties of probabilistic computations, and Markov chains in particular. These examples cannot be verified with existing approaches.

First, we verify a classic example of probabilistic non-interference which requires the reasoning about computations at different types. Second, in the context of Markov chains, we verify an example about stochastic dominance which exercises our more general rule for proving the existence of couplings modelled by expressions of different types. Finally, we verify an example involving shift relations in an infinite computation. This style of reasoning is motivated by "shift" couplings in Markov chains. In contrast to a standard coupling, which relates the states of two Markov chains at the same time t, a shift coupling relates the states of two Markov chains at possibly different times. Our specific example relates a standard random walk (described earlier) to a variant called a lazy random walk; the verification requires relating the state of standard random walk at time t to the state of the lazy random walk at time $2t$. We note that this kind of reasoning is impossible with conventional relational proof rules even in a non-probabilistic setting. Therefore, we provide a novel family of proof rules for reasoning about shift relations. At a high level, the rules combine a careful treatment of the later and constant modalities with a refined treatment of fixpoint operators, allowing us to relate different iterates of function bodies.

Summary of Contributions

With the aim of providing a general framework for programming and reasoning about Markov chains, the three main contributions of this work are:

1. A probabilistic extension of the guarded λ-calculus, that enables the definition of Markov chains as discrete probability distributions over streams.
2. A relational logic based on coupling to reason in a syntax-directed manner about (relational) properties of Markov chains. This logic supports reasoning

about programs that have different types and structures. Additionally, this logic uses results from the coupling literature to achieve greater expressivity than previous systems.
3. An extension of the relational logic that allows to relate the states of two streams at possibly different times. This extension supports reasoning principles, such as shift couplings, that escape conventional relational logics.

Omitted technical details can be found in the full version of the paper with appendix at https://arxiv.org/abs/1802.09787.

2 Mathematical Preliminaries

This section reviews the definition of discrete probability sub-distributions and introduces mathematical couplings.

Definition 1 (Discrete probability distribution). *Let C be a discrete (i.e., finite or countable) set. A (total) distribution over C is a function $\mu : C \to [0, 1]$ such that $\sum_{x \in C} \mu(x) = 1$. The support of a distribution μ is the set of points with non-zero probability,* $\mathsf{supp}\ \mu \triangleq \{x \in C \mid \mu(x) > 0\}$. *We denote the set of distributions over C as $\mathsf{D}(C)$. Given a subset $E \subseteq C$, the probability of sampling from μ a point in E is denoted $\Pr_{x \leftarrow \mu}[x \in E]$, and is equal to $\sum_{x \in E} \mu(x)$.*

Definition 2 (Marginals). *Let μ be a distribution over a product space $C_1 \times C_2$. The first (second) marginal of μ is another distribution $\mathsf{D}(\pi_1)(\mu)$ ($\mathsf{D}(\pi_2)(\mu)$) over C_1 (C_2) defined as:*

$$\mathsf{D}(\pi_1)(\mu)(x) = \sum_{y \in C_2} \mu(x, y) \qquad \left(\mathsf{D}(\pi_2)(\mu)(y) = \sum_{x \in C_1} \mu(x, y)\right)$$

Probabilistic Couplings. Probabilistic couplings are a fundamental tool in the analysis of Markov chains. When analyzing a relation between two probability distributions it is sometimes useful to consider instead a distribution over the product space that somehow "couples" the randomness in a convenient manner.

Consider for instance the case of the following Markov chain, which counts the total amount of tails observed when tossing repeatedly a biased coin with probability of tails p:

$$n_0 = 0 \qquad n_{i+1} = \begin{cases} n_i + 1 \text{ with probability } p \\ n_i \text{ with probability } (1 - p) \end{cases}$$

If we have two biased coins with probabilities of tails p and q with $p \leq q$ and we respectively observe $\{n_i\}$ and $\{m_i\}$ we would expect that, in some sense, $n_i \leq m_i$ should hold for all i (this property is known as stochastic dominance). A formal proof of this fact using elementary tools from probability theory would require to compute the cumulative distribution functions for n_i and m_i and then to compare them. The coupling method reduces this proof to showing a way to pair the coin flips so that if the first coin shows tails, so does the second coin.

We now review the definition of couplings and state relevant properties.

Definition 3 (Couplings). *Let $\mu_1 \in \mathsf{D}(C_1)$ and $\mu_2 \in \mathsf{D}(C_2)$, and $R \subseteq C_1 \times C_2$.*

- *A distribution $\mu \in \mathsf{D}(C_1 \times C_2)$ is a coupling for μ_1 and μ_2 iff its first and second marginals coincide with μ_1 and μ_2 respectively, i.e. $\mathsf{D}(\pi_1)(\mu) = \mu_1$ and $\mathsf{D}(\pi_2)(\mu) = \mu_2$.*
- *A distribution $\mu \in \mathsf{D}(C_1 \times C_2)$ is a R-coupling for μ_1 and μ_2 if it is a coupling for μ_1 and μ_2 and, moreover, $\Pr_{(x_1,x_2)\leftarrow\mu}[R\ x_1\ x_2] = 1$, i.e., if the support of the distribution μ is included in R.*

Moreover, we write $\diamond_{\mu_1,\mu_2}.R$ iff there exists a R-coupling for μ_1 and μ_2.

Couplings always exist. For instance, the product distribution of two distributions is always a coupling. Going back to the example about the two coins, it can be proven by computation that the following is a coupling that lifts the less-or-equal relation (0 indicating heads and 1 indicating tails):

$$\begin{cases} (0,0) \text{ w/ prob } (1-q) & (0,1) \text{ w/ prob } (q-p) \\ (1,0) \text{ w/ prob } 0 & (1,1) \text{ w/ prob } p \end{cases}$$

The following theorem in [7] gives a necessary and sufficient condition for the existence of R-couplings between two distributions. The theorem is remarkable in the sense that it proves an equivalence between an existential property (namely the existence of a particular coupling) and a universal property (checking, for each event, an inequality between probabilities).

Theorem 1 (Strassen's theorem). *Consider $\mu_1 \in \mathsf{D}(C_1)$ and $\mu_2 \in \mathsf{D}(C_2)$, and $R \subseteq C_1 \times C_2$. Then $\diamond_{\mu_1,\mu_2}.R$ iff for every $X \subseteq C_1$, $\Pr_{x_1\leftarrow\mu_1}[x_1 \in X] \leq \Pr_{x_2\leftarrow\mu_2}[x_2 \in R(X)]$, where $R(X)$ is the image of X under R, i.e. $R(X) = \{y \in C_2 \mid \exists x \in X.\ R\ x\ y\}$.*

An important property of couplings is closure under sequential composition.

Lemma 1 (Sequential composition couplings). *Let $\mu_1 \in \mathsf{D}(C_1)$, $\mu_2 \in \mathsf{D}(C_2)$, $M_1 : C_1 \to \mathsf{D}(D_1)$ and $M_2 : C_2 \to \mathsf{D}(D_2)$. Moreover, let $R \subseteq C_1 \times C_2$ and $S \subseteq D_1 \times D_2$. Assume:* (1) *$\diamond_{\mu_1,\mu_2}.R$; and* (2) *for every $x_1 \in C_1$ and $x_2 \in C_2$ such that $R\ x_1\ x_2$, we have $\diamond_{M_1(x_1),M_2(x_2)}.S$. Then $\diamond_{(\mathsf{bind}\ \mu_1\ M_1),(\mathsf{bind}\ \mu_2\ M_2)}.S$, where $\mathsf{bind}\ \mu\ M$ is defined as*

$$(\mathsf{bind}\ \mu\ M)(y) = \sum_x \mu(x) \cdot M(x)(y)$$

We conclude this section with the following lemma, which follows from Strassen's theorem:

Lemma 2 (Fundamental lemma of couplings). *Let $R \subseteq C_1 \times C_2$, $E_1 \subseteq C_1$ and $E_2 \subseteq C_2$ such that for every $x_1 \in E_1$ and $x_2 \in C_2$, $R\ x_1\ x_2$ implies $x_2 \in E_2$, i.e. $R(E_1) \subseteq E_2$. Moreover, let $\mu_1 \in \mathsf{D}(C_1)$ and $\mu_2 \in \mathsf{D}(C_2)$ such that $\diamond_{\mu_1,\mu_2}.R$. Then*

$$\Pr_{x_1\leftarrow\mu_1}[x_1 \in E_1] \leq \Pr_{x_2\leftarrow\mu_2}[x_2 \in E_2]$$

This lemma can be used to prove probabilistic inequalities from the existence of suitable couplings:

Corollary 1. *Let $\mu_1, \mu_2 \in \mathsf{D}(C)$:*

1. *If $\diamond_{\mu_1,\mu_2}.(=)$, then for all $x \in C$, $\mu_1(x) = \mu_2(x)$.*
2. *If $C = \mathbb{N}$ and $\diamond_{\mu_1,\mu_2}.(\geq)$, then for all $n \in \mathbb{N}$,* $\Pr_{x \leftarrow \mu_1}[x \geq n] \geq \Pr_{x \leftarrow \mu_2}[x \geq n]$

In the example at the beginning of the section, the property we want to prove is precisely that, for every k and i, the following holds:

$$\Pr_{x_1 \leftarrow n_i}[x_1 \geq k] \leq \Pr_{x_2 \leftarrow m_i}[x_2 \geq k]$$

Since we have a $\leq$-coupling, this proof is immediate. This example is formalized in Subsect. 3.3.

3 Overview of the System

In this section we give a high-level overview of our system, with the details on Sects. 4, 5 and 6. We start by presenting the base logic, and then we show how to extend it with probabilities and how to build a relational reasoning system on top of it.

3.1 Base Logic: Guarded Higher-Order Logic

Our starting point is the Guarded Higher-Order Logic [1] (Guarded HOL) inspired by the topos of trees. In addition to the usual constructs of HOL to reason about lambda terms, this logic features the $\triangleright$ and $\square$ modalities to reason about infinite terms, in particular streams. The $\triangleright$ modality is used to reason about objects that will be available in the future, such as tails of streams. For instance, suppose we want to define an $\mathrm{All}(s, \phi)$ predicate, expressing that all elements of a stream $s \equiv n{::}xs$ satisfy a property ϕ. This can be axiomatized as follows:

$$\forall (xs : \triangleright \mathrm{Str}_{\mathbb{N}})(n : \mathbb{N}).\phi\ n \Rightarrow \triangleright [s \leftarrow xs] \,.\, \mathrm{All}(s, x.\phi) \Rightarrow \mathrm{All}(n{::}xs, x.\phi)$$

We use $x.\phi$ to denote that the formula ϕ depends on a free variable x, which will get replaced by the first argument of All. We have two antecedents. The first one states that the head n satisfies ϕ. The second one, $\triangleright [s \leftarrow xs] \,.\, \mathrm{All}(s, x.\phi)$, states that all elements of xs satisfy ϕ. Formally, xs is the tail of the stream and will be available in the future, so it has type $\triangleright \mathrm{Str}_{\mathbb{N}}$. The *delayed substitution* $\triangleright[s \leftarrow xs]$ replaces s of type $\mathrm{Str}_{\mathbb{N}}$ with xs of type $\triangleright \mathrm{Str}_{\mathbb{N}}$ inside All and shifts the whole formula one step into the future. In other words, $\triangleright [s \leftarrow xs] \,.\, \mathrm{All}(s, x.\phi)$ states that $\mathrm{All}(-, x.\phi)$ will be satisfied by xs in the future, once it is available.

3.2 A System for Relational Reasoning

When proving relational properties it is often convenient to build proofs guided by the syntactic structure of the two expressions to be related. This style of reasoning is particularly appealing when the two expressions have the same structure and control-flow, and is appealingly close to the traditional style of reasoning supported by refinement types. At the same time, a strict adherence to the syntax-directed discipline is detrimental to the expressiveness of the system; for instance, it makes it difficult or even impossible to reason about structurally dissimilar terms. To achieve the best of both worlds, we present a relational proof system built on top of Guarded HOL, which we call Guarded RHOL. Judgements have the shape:

$$\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash t_1 : A_1 \sim t_2 : A_2 \mid \phi$$

where ϕ is a logical formula that may contain two distinguished variables $\mathbf{r}_1$ and $\mathbf{r}_2$ that respectively represent the expressions t_1 and t_2. This judgement subsumes two typing judgements on t_1 and t_2 and a relation ϕ on these two expressions. However, this form of judgement does not tie the logical property to the type of the expressions, and is key to achieving flexibility while supporting syntax-directed proofs whenever needed. The proof system combines rules of two different flavours: two-sided rules, which relate expressions with the same top-level constructs, and one-sided rules, which operate on a single expression.

We then extend Guarded HOL with a modality $\diamond$ that lifts assertions over discrete types C_1 and C_2 to assertions over $\mathsf{D}(C_1)$ and $\mathsf{D}(C_2)$. Concretely, we define for every assertion ϕ, variables x_1 and x_2 of type C_1 and C_2 respectively, and expressions t_1 and t_2 of type $\mathsf{D}(C_1)$ and $\mathsf{D}(C_2)$ respectively, the modal assertion $\diamond_{[x_1 \leftarrow t_1, x_2 \leftarrow t_2]}\phi$ which holds iff the interpretations of t_1 and t_2 are related by the probabilistic lifting of the interpretation of ϕ. We call this new logic Probabilistic Guarded HOL.

We accordingly extend the relational proof system to support reasoning about probabilistic expressions by adding judgements of the form:

$$\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash t_1 : \mathsf{D}(C_1) \sim t_2 : \mathsf{D}(C_2) \mid \diamond_{[x_1 \leftarrow \mathbf{r}_1, x_2 \leftarrow \mathbf{r}_2]}\phi$$

expressing that t_1 and t_2 are distributions related by a ϕ-coupling. We call this proof system Probabilistic Guarded RHOL. These judgements can be built by using the following rule, that lifts relational judgements over discrete types C_1 and C_2 to judgements over distribution types $\mathsf{D}(C_1)$ and $\mathsf{D}(C_2)$ when the premises of Strassen's theorem are satisfied.

$$\frac{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \forall X_1 \subseteq C_1 .\, \Pr_{y_1 \leftarrow t_1}[y_1 \in X_1] \leq \Pr_{y_2 \leftarrow t_2}[\exists y_1 \in X_1 . \phi]}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash t_1 : \mathsf{D}(C_1) \sim t_2 : \mathsf{D}(C_2) \mid \diamond_{[y_1 \leftarrow \mathbf{r}_1, y_2 \leftarrow \mathbf{r}_2]}\phi} \ \textsf{COUPLING}$$

Recall that (discrete time) Markov chains are "memoryless" probabilistic processes, whose specification is given by a (discrete) set C of states, an initial state s_0 and a probabilistic transition function $\mathsf{step} : C \to \mathsf{D}(C)$, where $\mathsf{D}(S)$ represents the set of discrete distributions over C. As explained in the introduction, a convenient modelling of Markov chains is by means of probabilistic streams, i.e. to model a Markov chain as an element of $\mathsf{D}(\mathrm{Str}_S)$, where S is its underlying state space. To model Markov chains, we introduce a markov operator with type $C \to (C \to \mathsf{D}(C)) \to \mathsf{D}(\mathrm{Str}_C)$ that, given an initial state and a transition function, returns a Markov chain. We can reason about Markov chains by the [Markov] rule (the context, omitted, does not change):

$$\frac{\begin{array}{c}\vdash t_1 : C_1 \sim t_2 : C_2 \mid \phi \\ \vdash h_1 : C_1 \to \mathsf{D}(C_1) \sim h_2 : C_2 \to \mathsf{D}(C_2) \mid \psi_3 \\ \vdash \psi_4\end{array}}{\vdash \mathrm{markov}(t_1, h_1) : \mathsf{D}(\mathrm{Str}_{D_1}) \sim \mathrm{markov}(t_2, h_2) : \mathsf{D}(\mathrm{Str}_{D_2}) \mid \diamond_{\left[\substack{y_1 \leftarrow \mathbf{r}_1 \\ y_2 \leftarrow \mathbf{r}_2}\right]} \phi'}\ \mathsf{Markov}$$

$$\text{where} \begin{cases} \psi_3 \equiv \forall x_1 x_2. \phi[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \Rightarrow \diamond_{[y_1 \leftarrow \mathbf{r}_1\ x_1, y_2 \leftarrow \mathbf{r}_2\ x_2]} \phi[y_1/\mathbf{r}_1][y_2/\mathbf{r}_2] \\ \psi_4 \equiv \forall x_1\ x_2\ xs_1\ xs_2. \phi[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \Rightarrow \triangleright[y_1 \leftarrow xs_1, y_2 \leftarrow xs_2].\phi' \Rightarrow \\ \qquad\qquad \phi'[x_1{::}xs_1/y_1][x_2{::}xs_2/y_2] \end{cases}$$

Informally, the rule stipulates the existence of an invariant ϕ over states. The first premise insists that the invariant hold on the initial states, the condition ψ_3 states that the transition functions preserve the invariant, and ψ_4 states that the invariant ϕ over pairs of states can be lifted to a stream property ϕ'.

Other rules of the logic are given in Fig. 1. The language construct munit creates a point distribution whose entire mass is at its argument. Accordingly, the [UNIT] rule creates a straightforward coupling. The [MLET] rule internalizes sequential composition of couplings (Lemma 1) into the proof system. The construct $\mathsf{let}\ x = t\ \mathsf{in}\ t'$ composes a distribution t with a probabilistic computation t' with one free variable x by sampling x from t and running t'. The [$\mathsf{MLET\text{-}L}$] rule supports one-sided reasoning about $\mathsf{let}\ x = t\ \mathsf{in}\ t'$ and relies on the fact that couplings are closed under convex combinations. Note that one premise of the rule uses a unary judgement, with a non-relational modality $\diamond_{[x \leftarrow \mathbf{r}]}\phi$ whose informal meaning is that ϕ holds with probability 1 in the distribution $\mathbf{r}$.

The following table summarizes the different base logics we consider, the relational systems we build on top of them, including the ones presented in [2], and the equivalences between both sides:

Relational logic		Base logic
RHOL [2] $\Gamma \mid \Psi \vdash t_1 \sim t_2 \mid \phi$	$\overset{[2]}{\Longleftrightarrow}$	HOL [2] $\Gamma \mid \Psi \vdash \phi[t_1/\mathbf{r}_1][t_2/\mathbf{r}_2]$
Guarded RHOL §6 $\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash t_1 \sim t_2 \mid \phi$	$\overset{\text{Thm 3}}{\Longleftrightarrow}$	Guarded HOL [1] $\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \phi[t_1/\mathbf{r}_1][t_2/\mathbf{r}_2]$
Probabilistic Guarded RHOL §6 $\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash t_1 \sim t_2 \mid \diamond_{[y_1 \leftarrow \mathbf{r}_1, y_2 \leftarrow \mathbf{r}_2]}.\phi$	$\overset{\text{Thm 3}}{\Longleftrightarrow}$	Probabilistic Guarded HOL §5 $\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \diamond_{[y_1 \leftarrow t_1, y_2 \leftarrow t_2]}.\phi$

3.3 Examples

We formalize elementary examples from the literature on security and Markov chains. None of these examples can be verified in prior systems. Uniformity of

$$\frac{\varDelta \mid \varSigma \mid \varGamma \mid \varPsi \vdash t_1 : C_1 \sim t_2 : C_2 \mid \phi[\mathbf{r}_1/x_1, \mathbf{r}_2/x_2]}{\varDelta \mid \varSigma \mid \varGamma \mid \varPsi \vdash \mathsf{munit}(t_1) : \mathsf{D}(C_1) \sim \mathsf{munit}(t_2) : \mathsf{D}(C_2) \mid \diamond_{[x_1 \leftarrow \mathbf{r}_1, x_2 \leftarrow \mathbf{r}_2]}\phi}\ \mathsf{UNIT}$$

$$\frac{\begin{array}{c}\varDelta \mid \varSigma \mid \varGamma \mid \varPsi \vdash t_1 : \mathsf{D}(C_1) \sim t_2 : \mathsf{D}(C_2) \mid \diamond_{[x_1 \leftarrow \mathbf{r}_1, x_2 \leftarrow \mathbf{r}_2]}\phi \\ \varDelta \mid \varSigma \mid \varGamma, x_1 : C_1, x_2 : C_2 \mid \varPsi, \phi \vdash t_1' : \mathsf{D}(D_1) \sim t_2' : \mathsf{D}(D_2) \mid \diamond_{[y_1 \leftarrow \mathbf{r}_1, y_2 \leftarrow \mathbf{r}_2]}\psi\end{array}}{\varDelta \mid \varSigma \mid \varGamma \mid \varPsi \vdash \mathsf{let}\ x_1 = t_1\ \mathsf{in}\ t_1' : \mathsf{D}(D_1) \sim \mathsf{let}\ x_2 = t_2\ \mathsf{in}\ t_2' : \mathsf{D}(D_2) \mid \diamond_{\left[\substack{y_1 \leftarrow \mathbf{r}_1 \\ y_2 \leftarrow \mathbf{r}_2}\right]}\psi}\ \mathsf{MLET}$$

$$\frac{\begin{array}{c}\varDelta \mid \varSigma \mid \varGamma \mid \varPsi \vdash t_1 : \mathsf{D}(C_1) \mid \diamond_{[x \leftarrow \mathbf{r}]}\phi \\ \varDelta \mid \varSigma \mid \varGamma, x_1 : C_1 \mid \varPsi, \phi \vdash t_1' : \mathsf{D}(D_1) \sim t_2' : \mathsf{D}(D_2) \mid \diamond_{[y_1 \leftarrow \mathbf{r}_1, y_2 \leftarrow \mathbf{r}_2]}\psi\end{array}}{\varDelta \mid \varSigma \mid \varGamma \mid \varPsi \vdash \mathsf{let}\ x_1 = t_1\ \mathsf{in}\ t_1' : \mathsf{D}(D_1) \sim t_2' : \mathsf{D}(D_2) \mid \diamond_{[y_1 \leftarrow \mathbf{r}_1, y_2 \leftarrow \mathbf{r}_2]}\psi}\ \mathsf{MLET-L}$$

Fig. 1. Proof rules for probabilistic constructs

one-time pad and lumping of *random walks* cannot even be stated in prior systems because the two related expressions in these examples have different types. The *random walk vs lazy random walk* (shift coupling) cannot be proved in prior systems because it requires either asynchronous reasoning or code rewriting. Finally, the *biased coin example* (stochastic dominance) cannot be proved in prior work because it requires Strassen's formulation of the existence of coupling (rather than a bijection-based formulation) or code rewriting. We give additional details below.

One-Time Pad/Probabilistic Non-interference. Non-interference [8] is a baseline information flow policy that is often used to model confidentiality of computations. In its simplest form, non-interference distinguishes between public (or low) and private (or high) variables and expressions, and requires that the result of a public expression not depend on the value of its private parameters. This definition naturally extends to probabilistic expressions, except that in this case the evaluation of an expression yields a distribution rather than a value. There are deep connections between probabilistic non-interference and several notions of (information-theoretic) security from cryptography. In this paragraph, we illustrate different flavours of security properties for one-time pad encryption. Similar reasoning can be carried out for proving (passive) security of secure multiparty computation algorithms in the 3-party or multi-party setting [9,10].

One-time pad is a perfectly secure symmetric encryption scheme. Its space of plaintexts, ciphertexts and keys is the set $\{0,1\}^\ell$—fixed-length bitstrings of size ℓ. The encryption algorithm is parametrized by a key k—sampled uniformly over the set of bitstrings $\{0,1\}^\ell$—and maps every plaintext m to the ciphertext $c = k \oplus m$, where the operator $\oplus$ denotes bitwise exclusive-or on bitstrings. We let otp denote the expression $\lambda m.\mathsf{let}\ k = \mathcal{U}_{\{0,1\}^\ell}\ \mathsf{in}\ \mathsf{munit}(k \oplus m)$, where $\mathcal{U}_X$ is the uniform distribution over a finite set X.

One-time pad achieves perfect security, i.e. the distributions of ciphertexts is independent of the plaintext. Perfect security can be captured as a probabilistic non-interference property:

$$\vdash \mathsf{otp} : \{0,1\}^\ell \to \mathsf{D}(\{0,1\}^\ell) \sim \mathsf{otp} : \{0,1\}^\ell \to \mathsf{D}(\{0,1\}^\ell) \mid \forall m_1 m_2 . \mathbf{r}_1\, m_1 \overset{\diamond}{=} \mathbf{r}_2\, m_2$$

where $e_1 \overset{\diamond}{=} e_2$ is used as a shorthand for $\diamond_{[y_1 \leftarrow e_1, y_2 \leftarrow e_2]} y_1 = y_2$. The crux of the proof is to establish

$$m_1, m_2 : \{0,1\}^\ell \vdash \mathcal{U}_{\{0,1\}^\ell} : \mathsf{D}(\{0,1\}^\ell) \sim \mathcal{U}_{\{0,1\}^\ell} : \mathsf{D}(\{0,1\}^\ell) \mid \mathbf{r}_1 \oplus m_2 \overset{\diamond}{=} \mathbf{r}_2 \oplus m_1$$

using the [COUPLING] rule. It suffices to observe that the assertion induces a bijection, so the image of an arbitrary set X under the relation has the same cardinality as X, and hence their probabilities w.r.t. the uniform distributions are equal. One can then conclude the proof by applying the rules for monadic sequenciation ([MLET]) and abstraction (rule [ABS] in appendix), using algebraic properties of $\oplus$.

Interestingly, one can prove a stronger property: rather than proving that the ciphertext is independent of the plaintext, one can prove that the distribution of ciphertexts is uniform. This is captured by the following judgement:

$$c_1, c_2 : \{0,1\}^\ell \vdash \mathsf{otp} : \{0,1\}^\ell \to \mathsf{D}(\{0,1\}^\ell) \sim \mathsf{otp} : \{0,1\}^\ell \to \mathsf{D}(\{0,1\}^\ell) \mid \psi$$

where $\psi \triangleq \forall m_1\, m_2 . m_1 = m_2 \Rightarrow \diamond_{[y_1 \leftarrow \mathbf{r}_1\ m_1, y_2 \leftarrow \mathbf{r}_2\ m_2]} y_1 = c_1 \Leftrightarrow y_2 = c_2$. This style of modelling uniformity as a relational property is inspired from [11]. The proof is similar to the previous one and omitted. However, it is arguably more natural to model uniformity of the distribution of ciphertexts by the judgement:

$$\vdash \mathsf{otp} : \{0,1\}^\ell \to \mathsf{D}(\{0,1\}^\ell) \sim \mathcal{U}_{\{0,1\}^\ell} : \mathsf{D}(\{0,1\}^\ell) \mid \forall m.\ \mathbf{r}_1\ m \overset{\diamond}{=} \mathbf{r}_2$$

This judgement is closer to the simulation-based notion of security that is used pervasively in cryptography, and notably in Universal Composability [12]. Specifically, the statement captures the fact that the one-time pad algorithm can be simulated without access to the message. It is interesting to note that the judgement above (and more generally simulation-based security) could not be expressed in prior works, since the two expressions of the judgement have different types—note that in this specific case, the right expression is a distribution but in the general case the right expression will also be a function, and its domain will be a projection of the domain of the left expression.

The proof proceeds as follows. First, we prove

$$\vdash \mathcal{U}_{\{0,1\}^\ell} \sim \mathcal{U}_{\{0,1\}^\ell} \mid \forall m.\ \diamond_{[y_1 \leftarrow \mathbf{r}_1, y_2 \leftarrow \mathbf{r}_2]}\, y_1 \oplus m = y_2$$

using the [COUPLING] rule. Then, we apply the [MLET] rule to obtain

$$\vdash \begin{array}{l}\mathsf{let}\ k = \mathcal{U}_{\{0,1\}^\ell}\ \mathsf{in} \\ \mathsf{munit}(k \oplus m)\end{array} \sim \begin{array}{l}\mathsf{let}\ k = \mathcal{U}_{\{0,1\}^\ell}\ \mathsf{in} \\ \mathsf{munit}(k)\end{array} \mid \diamond_{[y_1 \leftarrow \mathbf{r}_1, y_2 \leftarrow \mathbf{r}_2]} y_1 = y_2$$

We have $\mathsf{let}\ k = \mathcal{U}_{\{0,1\}^\ell}\ \mathsf{in}\ \mathsf{munit}(k) \equiv \mathcal{U}_{\{0,1\}^\ell}$; hence by equivalence (rule [Equiv] in appendix), this entails

$$\vdash \mathsf{let}\ k = \mathcal{U}_{\{0,1\}^\ell}\ \mathsf{in}\ \mathsf{munit}(k \oplus m) \sim \mathcal{U}_{\{0,1\}^\ell} \mid \diamond_{[y_1 \leftarrow \mathbf{r}_1, y_2 \leftarrow \mathbf{r}_2]} y_1 = y_2$$

We conclude by applying the one-sided rule for abstraction.

Stochastic Dominance. Stochastic dominance defines a partial order between random variables whose underlying set is itself a partial order; it has many different applications in statistical biology (e.g. in the analysis of the birth-and-death processes), statistical physics (e.g. in percolation theory), and economics. First-order stochastic dominance, which we define below, is also an important application of probabilistic couplings. We demonstrate how to use our proof system for proving (first-order) stochastic dominance for a simple Markov process which samples biased coins. While the example is elementary, the proof method extends to more complex examples of stochastic dominance, and illustrates the benefits of Strassen's formulation of the coupling rule over alternative formulations stipulating the existence of bijections (explained later).

We start by recalling the definition of (first-order) stochastic dominance for the $\mathbb{N}$-valued case. The definition extends to arbitrary partial orders.

Definition 4 (Stochastic dominance). *Let $\mu_1, \mu_2 \in \mathsf{D}(\mathbb{N})$. We say that μ_2 stochastically dominates μ_1, written $\mu_1 \leq_{\text{SD}} \mu_2$, iff for every $n \in \mathbb{N}$,*

$$\Pr_{x \leftarrow \mu_1}[x \geq n] \leq \Pr_{x \leftarrow \mu_2}[x \geq n]$$

The following result, equivalent to Corollary 1, characterizes stochastic dominance using probabilistic couplings.

Proposition 1. *Let $\mu_1, \mu_2 \in \mathsf{D}(\mathbb{N})$. Then $\mu_1 \leq_{\text{SD}} \mu_2$ iff $\diamond_{\mu_1,\mu_2}.(\leq)$.*

We now turn to the definition of the Markov chain. For $p \in [0,1]$, we consider the parametric $\mathbb{N}$-valued Markov chain $\mathsf{coins} \triangleq \mathsf{markov}(0, h)$, with initial state 0 and (parametric) step function:

$$h \triangleq \lambda x.\mathsf{let}\ b = \mathcal{B}(p)\ \mathsf{in}\ \mathsf{munit}(x + b)$$

where, for $p \in [0,1]$, $\mathcal{B}(p)$ is the Bernoulli distribution on $\{0,1\}$ with probability p for 1 and $1-p$ for 0. Our goal is to establish that coins is monotonic, i.e. for every $p_1, p_2 \in [0,1]$, $p_1 \leq p_2$ implies $\mathsf{coins}\ p_1 \leq_{\text{SD}} \mathsf{coins}\ p_2$. We formalize this statement as

$$\vdash \mathsf{coins} : [0,1] \to \mathsf{D}(\text{Str}_{\mathbb{N}}) \sim \mathsf{coins} : [0,1] \to \mathsf{D}(\text{Str}_{\mathbb{N}}) \mid \psi$$

where $\psi \triangleq \forall p_1, p_2 . p_1 \leq p_2 \Rightarrow \diamond_{[y_1 \leftarrow \mathbf{r}_1, y_2 \leftarrow \mathbf{r}_2]} \text{All}(y_1, y_2, z_1.z_2.z_1 \leq z_2)$. The crux of the proof is to establish stochastic dominance for the Bernoulli distribution:

$$p_1 : [0,1], p_2 : [0,1] \mid p_1 \leq p_2 \vdash \mathcal{B}(p_1) : \mathsf{D}(\mathbb{N}) \sim \mathcal{B}(p_2) : \mathsf{D}(\mathbb{N}) \mid \mathbf{r}_1 \overset{\diamond}{\leq} \mathbf{r}_2$$

where we use $e_1 \overset{\diamond}{\leq} e_2$ as shorthand for $\diamond_{[y_1 \leftarrow e_1, y_2 \leftarrow e_2]} y_1 \leq y_2$. This is proved directly by the [$\mathsf{COUPLING}$] rule and checking by simple calculations that the premise of the rule is valid.

We briefly explain how to conclude the proof. Let h_1 and h_2 be the step functions for p_1 and p_2 respectively. It is clear from the above that (context omitted):

$$x_1 \leq x_2 \vdash h_1\ x_1 : \mathsf{D}(\mathbb{B}) \sim h_2\ x_2 : \mathsf{D}(\mathbb{B}) \mid \diamond_{[y_1 \leftarrow \mathbf{r}_1, y_2 \leftarrow \mathbf{r}_2]} . y_1 \leq y_2$$

and by the definition of All:

$$x_1 \leq x_2 \Rightarrow \mathrm{All}(xs_1, xs_2, z_1.z_2.z_1 \leq z_2) \Rightarrow \mathrm{All}(x_1{::}\triangleright xs_1, x_2{::}\triangleright xs_2, z_1.z_2.z_1 \leq z_2)$$

So, we can conclude by applying the [Markov] rule.

It is instructive to compare our proof with prior formalizations, and in particular with the proof in [5]. Their proof is carried out in the pRHL logic, whose [COUPLING] rule is based on the existence of a bijection that satisfies some property, rather than on our formalization based on Strassen's Theorem. Their rule is motivated by applications in cryptography, and works well for many examples, but is inconvenient for our example at hand, which involves non-uniform probabilities. Indeed, their proof is based on code rewriting, and is done in two steps. First, they prove equivalence between sampling and returning x_1 from $\mathcal{B}(p_1)$; and sampling z_1 from $\mathcal{B}(p_2)$, z_2 from $\mathcal{B}({}^{p_1}/_{p_2})$ and returning $z = z_1 \wedge z_2$. Then, they find a coupling between z and $\mathcal{B}(p_2)$.

Shift Coupling: Random Walk vs Lazy Random Walk. The previous example is an instance of a lockstep coupling, in that it relates the k-th element of the first chain with the k-th element of the second chain. Many examples from the literature follow this lockstep pattern; however, it is not always possible to establish lockstep couplings. Shift couplings are a relaxation of lockstep couplings where we relate elements of the first and second chains without the requirement that their positions coincide.

We consider a simple example that motivates the use of shift couplings. Consider the random walk and lazy random walk (which, at each time step, either chooses to move or stay put), both defined as Markov chains over $\mathbb{Z}$. For simplicity, assume that both walks start at position 0. It is not immediate to find a coupling between the two walks, since the two walks necessarily get desynchronized whenever the lazy walk stays put. Instead, the trick is to consider a lazy random walk that moves two steps instead of one. The random walk and the lazy random walk of step 2 are defined by the step functions:

$$\begin{aligned} \mathrm{step} &\triangleq \lambda x.\mathsf{let}\ z = \mathcal{U}_{\{-1,1\}}\ \mathsf{in}\ \mathsf{munit}(z + x) \\ \mathrm{lstep2} &\triangleq \lambda x.\mathsf{let}\ z = \mathcal{U}_{\{-1,1\}}\ \mathsf{in\ let}\ b = \mathcal{U}_{\{0,1\}}\ \mathsf{in}\ \mathsf{munit}(x + 2 * z * b) \end{aligned}$$

After 2 iterations of step, the position has either changed two steps to the left or to the right, or has returned to the initial position, which is the same behaviour lstep2 has on every iteration. Therefore, the coupling we want to find should equate the elements at position $2i$ in step with the elements at position i in lstep2. The details on how to prove the existence of this coupling are in Sect. 6.

Lumped Coupling: Random Walks on 3 and 4 Dimensions. A Markov chain is *recurrent* if it has probability 1 of returning to its initial state, and *transient* otherwise. It is relatively easy to show that the random walk over $\mathbb{Z}$ is recurrent. One can also show that the random walk over $\mathbb{Z}^2$ is recurrent. However, the random walk over $\mathbb{Z}^3$ is transient.

For higher dimensions, we can use a coupling argument to prove transience. Specifically, we can define a coupling between a lazy random walk in n dimensions and a random walk in $n+m$ dimensions, and derive transience of the latter from transience of the former. We define the (lazy) random walks below, and sketch the coupling arguments.

Specifically, we show here the particular case of the transience of the 4-dimensional random walk from the transience of the 3-dimensional lazy random walk. We start by defining the stepping functions:

$$\begin{aligned} \mathrm{step}_4 : \mathbb{Z}^4 \to \mathsf{D}(\mathbb{Z}^4) &\triangleq \lambda z_1.\mathsf{let}\ x_1 = \mathcal{U}_{U_4}\ \mathsf{in}\ \mathsf{munit}(z_1 +_4 x_1) \\ \mathrm{lstep}_3 : \mathbb{Z}^3 \to \mathsf{D}(\mathbb{Z}^3) &\triangleq \lambda z_2.\mathsf{let}\ x_2 = \mathcal{U}_{U_3}\ \mathsf{in}\ \mathsf{let}\ b_2 = \mathcal{B}(3/4)\ \mathsf{in}\ \mathsf{munit}(z_2 +_3 b_2 * x_2) \end{aligned}$$

where $U_i = \{(\pm 1, 0, \dots 0), \dots, (0, \dots, 0, \pm 1)\}$ are the vectors of the basis of $\mathbb{Z}^i$ and their opposites. Then, the random walk of dimension 4 is modelled by $\mathrm{rwalk4} \triangleq \mathsf{markov}(0, \mathrm{step}_4)$, and the lazy walk of dimension 3 is modelled by $\mathrm{lwalk3} \triangleq \mathsf{markov}(0, \mathrm{step}_3)$. We want to prove:

$$\vdash \mathrm{rwalk4} : \mathsf{D}(\mathrm{Str}_{\mathbb{Z}^4}) \sim \mathrm{lwalk3} : \mathsf{D}(\mathrm{Str}_{\mathbb{Z}^3}) \mid \diamond_{\left[\substack{y_1 \leftarrow \mathbf{r}_1 \\ y_2 \leftarrow \mathbf{r}_2}\right]} \mathrm{All}(y_1, y_2, z_1.z_2.\, \mathrm{pr}^4_3(z_1) = z_2)$$

where $\mathrm{pr}^{n_2}_{n_1}$ denotes the standard projection from $\mathbb{Z}^{n_2}$ to $\mathbb{Z}^{n_1}$.

We apply the [Markov] rule. The only interesting premise requires proving that the transition function preserves the coupling:

$$p_2 = \mathrm{pr}^4_3(p_1) \vdash \mathrm{step}_4 \sim \mathrm{lstep}_3 \mid \forall x_1 x_2.x_2 = \mathrm{pr}^4_3(x_1) \Rightarrow \diamond_{\left[\substack{y_1 \leftarrow \mathbf{r}_1\ x_1 \\ y_2 \leftarrow \mathbf{r}_2\ x_2}\right]} \mathrm{pr}^4_3(y_1) = y_2$$

To prove this, we need to find the appropriate coupling, i.e., one that preserves the equality. The idea is that the step in $\mathbb{Z}^3$ must be the projection of the step in $\mathbb{Z}^4$. This corresponds to the following judgement:

$$\begin{array}{l} \lambda z_1.\ \mathsf{let}\ x_1 = \mathcal{U}_{U_4}\ \mathsf{in} \\ \qquad \mathsf{munit}(z_1 +_4 x_1) \end{array} \sim \begin{array}{l} \lambda z_2.\ \mathsf{let}\ x_2 = \mathcal{U}_{U_3}\ \mathsf{in} \\ \qquad \mathsf{let}\ b_2 = \mathcal{B}(3/4)\ \mathsf{in} \\ \qquad \mathsf{munit}(z_2 +_3 b_2 * x_2) \end{array} \left| \begin{array}{c} \forall z_1 z_2.\, \mathrm{pr}^4_3(z_1) = z_2 \Rightarrow \\ \mathrm{pr}^4_3(\mathbf{r}_1\ z_1) \overset{\diamond}{=} \mathbf{r}_2\ z_2 \end{array} \right.$$

which by simple equational reasoning is the same as

$$\begin{array}{l} \lambda z_1.\ \mathsf{let}\ x_1 = \mathcal{U}_{U_4}\ \mathsf{in} \\ \qquad \mathsf{munit}(z_1 +_4 x_1) \end{array} \sim \begin{array}{l} \lambda z_2.\ \mathsf{let}\ p_2 = \mathcal{U}_{U_3} \times \mathcal{B}(3/4)\ \mathsf{in} \\ \qquad \mathsf{munit}(z_2 +_3 \pi_1(p_2) * \pi_2(p_2)) \end{array} \left| \begin{array}{c} \forall z_1 z_2.\, \mathrm{pr}^4_3(z_1) = z_2 \Rightarrow \\ \mathrm{pr}^4_3(\mathbf{r}_1\ z_1) \overset{\diamond}{=} \mathbf{r}_2\ z_2 \end{array} \right.$$

We want to build a coupling such that if we sample $(0, 0, 0, 1)$ or $(0, 0, 0, -1)$ from $\mathcal{U}_{U_3}$, then we sample 0 from $\mathcal{B}(3/4)$, and otherwise if we sample $(x_1, x_2, x_3, 0)$ from $\mathcal{U}_{U_4}$, we sample (x_1, x_2, x_3) from U_3. Formally, we prove this with the [Coupling] rule. Given $X : U_4 \to \mathbb{B}$, by simple computation we show that:

$$\Pr_{z_1 \sim \mathcal{U}_{U_4}} [z_1 \in X] \le \Pr_{z_2 \sim \mathcal{U}_{U_3} \times \mathcal{B}(3/4)} [z_2 \in \{y \mid \exists x \in X.\mathsf{pr}^4_3(x) = \pi_1(y) * \pi_2(y)\}]$$

This concludes the proof. From the previous example, it follows that the lazy walk in 3 dimensions is transient, since the random walk in 3 dimensions is transient. By simple reasoning, we now conclude that the random walk in 4 dimensions is also transient.

4 Probabilistic Guarded Lambda Calculus

To ensure that a function on infinite datatypes is well-defined, one must check that it is *productive*. This means that any finite prefix of the output can be computed in finite time. For instance, consider the following function on streams:

$$\texttt{letrec bad } (\texttt{x} : \texttt{xs}) = \texttt{x} : \texttt{tail}(\texttt{bad xs})$$

This function is not productive since only the first element can be computed. We can argue this as follows: Suppose that the tail of a stream is available one unit of time after its head, and that `x:xs` is available at time 0. How much time does it take for `bad` to start outputting its tail? Assume it takes k units of time. This means that `tail(bad xs)` will be available at time $k+1$, since `xs` is only available at time 1. But `tail(bad xs)` is exactly the tail of `bad(x:xs)`, and this is a contradiction, since `x:xs` is available at time 0 and therefore the tail of `bad(x:xs)` should be available at time k. Therefore, the tail of `bad` will never be available.

The guarded lambda calculus solves the productivity problem by distinguishing at type level between data that is available now and data that will be available in the future, and restricting when fixpoints can be defined. Specifically, the guarded lambda calculus extends the usual simply typed lambda calculus with two modalities: $\triangleright$ (pronounced *later*) and $\square$ (*constant*). The later modality represents data that will be available one step in the future, and is introduced and removed by the term formers $\triangleright$ and prev respectively. This modality is used to guard recursive occurrences, so for the calculus to remain productive, we must restrict when it can be eliminated. This is achieved via the constant modality, which expresses that all the data is available at all times. In the remainder of this section we present a probabilistic extension of this calculus.

Syntax. Types of the calculus are defined by the grammar

$$A, B ::= b \mid \mathbb{N} \mid A \times B \mid A + B \mid A \to B \mid \mathrm{Str}_A \mid \square\, A \mid \triangleright A \mid \mathsf{D}(C)$$

where b ranges over a collection of base types. Str_A is the type of guarded streams of elements of type A. Formally, the type Str_A is isomorphic to $A \times \triangleright \mathrm{Str}_A$. This isomorphism gives a way to introduce streams with the function $(::) : A \to \triangleright \mathrm{Str}_A \to \mathrm{Str}_A$ and to eliminate them with the functions $\mathrm{hd} : \mathrm{Str}_A \to A$ and $\mathrm{tl} : \mathrm{Str}_A \to \triangleright \mathrm{Str}_A$. $\mathsf{D}(C)$ is the type of distributions over *discrete types* C. Discrete types are defined by the following grammar, where b_0 are discrete base types, e.g., $\mathbb{Z}$.

$$C, D ::= b_0 \mid \mathbb{N} \mid C \times D \mid C + D \mid \mathrm{Str}_C \mid \triangleright C.$$

Note that, in particular, arrow types are not discrete but streams are. This is due to the semantics of streams as sets of finite approximations, which we describe in the next subsection. Also note that $\square\,\mathrm{Str}_A$ is not discrete since it makes the full infinite streams available.

We also need to distinguish between arbitrary types A, B and constant types S, T, which are defined by the following grammar

$$S, T ::= b_C \mid \mathbb{N} \mid S \times T \mid S + T \mid S \to T \mid \Box\, A$$

where b_C is a collection of constant base types. Note in particular that for any type A the type $\Box\, A$ is constant.

The terms of the language t are defined by the following grammar

$$\begin{aligned} t ::= {} & x \mid c \mid 0 \mid St \mid \text{case } t \text{ of } 0 \mapsto t; S \mapsto t \mid \mu \mid \mathsf{munit}(t) \mid \mathsf{let}\ x = t\ \mathsf{in}\ t \\ & \mid \langle t, t\rangle \mid \pi_1 t \mid \pi_2 t \mid \text{inj}_1 t \mid \text{inj}_2 t \mid \text{case}\, t\, \text{of inj}_1 x.t; \text{inj}_2 y.t \mid \lambda x.t \mid t\, t \mid \text{fix } x.\ t \\ & \mid t{::}ts \mid \text{hd}\, t \mid \text{tl}\, t \mid \text{box } t \mid \text{letb } x \leftarrow t \text{ in } t \mid \text{letc } x \leftarrow t \text{ in } t \mid \triangleright\xi.t \mid \text{prev } t \end{aligned}$$

where ξ is a delayed substitution, a sequence of bindings $[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$. The terms c are constants corresponding to the base types used and $\mathsf{munit}(t)$ and $\mathsf{let}\ x = t\ \mathsf{in}\ t$ are the introduction and sequencing construct for probability distributions. The meta-variable μ stands for base distributions like $\mathcal{U}_C$ and $\mathcal{B}(p)$.

Delayed substitutions were introduced in [13] in a dependent type theory to be able to work with types dependent on terms of type $\triangleright A$. In the setting of a simple type theory, such as the one considered in this paper, delayed substitutions are equivalent to having the applicative structure [14] $\circledast$ for the $\triangleright$ modality. However, delayed substitutions extend uniformly to the level of propositions, and thus we choose to use them in this paper in place of the applicative structure.

Denotational Semantics. The meaning of terms is given by a denotational model in the category $\mathcal{S}$ of presheaves over ω, the first infinite ordinal. This category $\mathcal{S}$ is also known as the *topos of trees* [15]. In previous work [1], it was shown how to model most of the constructions of the guarded lambda calculus and its internal logic, with the notable exception of the probabilistic features. Below we give an elementary presentation of the semantics.

Informally, the idea behind the topos of trees is to represent (infinite) objects from their finite approximations, which we observe incrementally as time passes. Given an object x, we can consider a sequence $\{x_i\}$ of its finite approximations observable at time i. These are trivial for finite objects, such as a natural number, since for any number n, $n_i = n$ at every i. But for infinite objects such as streams, the ith approximation is the prefix of length $i + 1$.

Concretely, the category $\mathcal{S}$ consists of:

- Objects X: families of sets $\{X_i\}_{i\in\mathbb{N}}$ together with *restriction functions* $r_n^X : X_{n+1} \to X_n$. We will write simply r_n if X is clear from the context.
- Morphisms $X \to Y$: families of functions $\alpha_n : X_n \to Y_n$ commuting with restriction functions in the sense of $r_n^Y \circ \alpha_{n+1} = \alpha_n \circ r_n^X$.

The full interpretation of types of the calculus can be found in Fig. 8 in the appendix. The main points we want to highlight are:

- Streams over a type A are interpreted as sequences of finite prefixes of elements of A with the restriction functions of A:

$$[\![\text{Str}_A]\!] \triangleq [\![A]\!]_0 \times \{*\} \xleftarrow{r_0 \times !} [\![A]\!]_1 \times [\![\text{Str}_A]\!]_0 \xleftarrow{r_1 \times r_0 \times !} [\![A]\!]_2 \times [\![\text{Str}_A]\!]_1 \leftarrow \cdots$$

– Distributions over a discrete object C are defined as a sequence of distributions over each $[\![C]\!]_i$:

$$[\![\mathsf{D}(C)]\!] \triangleq \mathsf{D}([\![C]\!]_0) \xleftarrow{\mathsf{D}(r_0)} \mathsf{D}([\![C]\!]_1) \xleftarrow{\mathsf{D}(r_1)} \mathsf{D}([\![C]\!]_2) \xleftarrow{\mathsf{D}(r_2)} \ldots,$$

where $\mathsf{D}([\![C]\!]_i)$ is the set of (probability density) functions $\mu : [\![C]\!]_i \to [0,1]$ such that $\sum_{x \in X} \mu x = 1$, and $\mathsf{D}(r_i)$ adds the probability density of all the points in $[\![C]\!]_{i+1}$ that are sent by r_i to the same point in the $[\![C]\!]_i$. In other words, $\mathsf{D}(r_i)(\mu)(x) = \Pr_{y \leftarrow \mu}[r_i(y) = x]$

An important property of the interpretation is that discrete types are interpreted as objects X such that X_i is finite or countably infinite for every i. This allows us to define distributions on these objects without the need for measure theory. In particular, the type of guarded streams Str_A is discrete provided A is, which is clear from the interpretation of the type Str_A. Conceptually this holds because $[\![\mathrm{Str}_A]\!]_i$ is an approximation of real streams, consisting of only the first $i+1$ elements.

An object X of $\mathcal{S}$ is *constant* if all its restriction functions are bijections. Constant types are interpreted as constant objects of $\mathcal{S}$ and for a constant type A the objects $[\![\Box A]\!]$ and $[\![A]\!]$ are isomorphic in $\mathcal{S}$.

Typing Rules. Terms are typed under a dual context $\Delta \mid \Gamma$, where Γ is a usual context that binds variables to a type, and Δ is a constant context containing variables bound to types that are *constant*. The term letc $x \leftarrow u$ in t allows us to shift variables between constant and non-constant contexts. The typing rules can be found in Fig. 2.

The semantics of such a dual context $\Delta \mid \Gamma$ is given as the product of types in Δ and Γ, except that we implicitly add $\Box$ in front of every type in Δ. In the particular case when both contexts are empty, the semantics of the dual context correspond to the terminal object 1, which is the singleton set $\{*\}$ at each time.

The interpretation of the well-typed term $\Delta \mid \Gamma \vdash t : A$ is defined by induction on the typing derivation, and can be found in Fig. 9 in the appendix.

Applicative Structure of the Later Modality. As in previous work we can define the operator $\circledast$ satisfying the typing rule

$$\frac{\Delta \mid \Gamma \vdash t : \triangleright(A \to B) \qquad \Delta \mid \Gamma \vdash u : \triangleright A}{\Delta \mid \Gamma \vdash t \circledast u : \triangleright B}$$

and the equation $(\triangleright t) \circledast (\triangleright u) \equiv \triangleright(t\ u)$ as the term $t \circledast u \triangleq \triangleright\left[f \leftarrow t, x \leftarrow u\right].f\,x$.

Example: Modelling Markov Chains. As an application of $\circledast$ and an example of how to use guardedness and probabilities together, we now give the precise definition of the markov construct that we used to model Markov chains earlier:

$$\begin{aligned}
&\mathsf{markov} : C \to (C \to \mathsf{D}(C)) \to \mathsf{D}(\mathrm{Str}_C)\\
&\mathsf{markov} \triangleq \mathrm{fix}\ f.\ \lambda x.\lambda h.\\
&\qquad\qquad \mathsf{let}\ z = h\ x\ \mathsf{in}\ \mathsf{let}\ t = \mathrm{swap}_{\triangleright\mathsf{D}}^{\mathrm{Str}_C}(f \circledast \triangleright z \circledast \triangleright h)\ \mathsf{in}\ \mathsf{munit}(x{::}t)
\end{aligned}$$

$$\frac{x : A \in \Gamma}{\Delta \mid \Gamma \vdash x : A} \qquad \frac{x : A \in \Delta}{\Delta \mid \Gamma \vdash x : A} \qquad \frac{\Delta \mid \Gamma, x : A \vdash t : B}{\Delta \mid \Gamma \vdash \lambda x.t : A \to B}$$

$$\frac{\Delta \mid \Gamma \vdash t : A \to B \qquad \Delta \mid \Gamma \vdash u : A}{\Delta \mid \Gamma \vdash t\, u : B} \qquad \frac{\Delta \mid \Gamma, f : \triangleright A \vdash t : A}{\Delta \mid \Gamma \vdash \text{fix } f.\ t : A} \qquad \frac{\Delta \mid \cdot \vdash t : \triangleright A}{\Delta \mid \Gamma \vdash \text{prev } t : A}$$

$$\frac{\Delta \mid \cdot \vdash t : A}{\Delta \mid \Gamma \vdash \text{box } t : \square A} \qquad \frac{\Delta \mid \Gamma \vdash u : \square B \qquad \Delta, x : B \mid \Gamma \vdash t : A}{\Delta \mid \Gamma \vdash \text{letb } x \leftarrow u \text{ in } t : A}$$

$$\frac{\Delta \mid \Gamma \vdash u : B \qquad \Delta, x : B \mid \Gamma \vdash t : A \qquad B \text{ constant}}{\Delta \mid \Gamma \vdash \text{letc } x \leftarrow u \text{ in } t : A}$$

$$\frac{\Delta \mid \Gamma, x_1 : A_1, \cdots x_n : A_n \vdash t : A \qquad \Delta \mid \Gamma \vdash t_i : \triangleright A_i}{\Delta \mid \Gamma \vdash \triangleright [x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n] .t : \triangleright A} \qquad \frac{\Delta \mid \Gamma \vdash t : A \qquad A \text{ discrete}}{\Delta \mid \Gamma \vdash \mathsf{munit}(t) : \mathsf{D}(A)}$$

$$\frac{\Delta \mid \Gamma \vdash t : \mathsf{D}(A) \qquad \Delta \mid \Gamma, x : A \vdash u : \mathsf{D}(B)}{\Delta \mid \Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ u : \mathsf{D}(B)} \qquad \frac{\mu \text{ primitive distribution on type } A}{\Delta \mid \Gamma \vdash \mu : \mathsf{D}(A)}$$

Fig. 2. A selection of the typing rules of the guarded lambda calculus. The rules for products, sums, and natural numbers are standard.

The guardedness condition gives f the type $\triangleright(C \to (C \to \mathsf{D}(C)) \to \mathsf{D}(\mathrm{Str}_C))$ in the body of the fixpoint. Therefore, it needs to be applied functorially (via $\circledast$) to $\triangleright z$ and $\triangleright h$, which gives us a term of type $\triangleright \mathsf{D}(\mathrm{Str}_C)$. To complete the definition we need to build a term of type $\mathsf{D}(\triangleright \mathrm{Str}_C)$ and then sequence it with :: to build a term of type $\mathsf{D}(\mathrm{Str}_C)$. To achieve this, we use the primitive operator $\mathrm{swap}^{\mathsf{C}}_{\triangleright \mathsf{D}} : \triangleright \mathsf{D}(C) \to \mathsf{D}(\triangleright C)$, which witnesses the isomorphism between $\triangleright \mathsf{D}(C)$ and $\mathsf{D}(\triangleright C)$. For this isomorphism to exist, it is crucial that distributions be total (i.e., we cannot use subdistributions). Indeed, the denotation for $\triangleright \mathsf{D}(C)$ is the sequence $\{*\} \leftarrow \mathsf{D}(C_1) \leftarrow \mathsf{D}(C_2) \leftarrow \ldots$, while the denotation for $\mathsf{D}(\triangleright C)$ is the sequence $\mathsf{D}(\{*\}) \leftarrow \mathsf{D}(C_1) \leftarrow \mathsf{D}(C_2) \leftarrow \ldots$, and $\{*\}$ is isomorphic to $\mathsf{D}(\{*\})$ in Set only if D considers only total distributions.

5 Guarded Higher-Order Logic

We now introduce Guarded HOL (GHOL), which is a higher-order logic to reason about terms of the guarded lambda calculus. The logic is essentially that of [1], but presented with the dual context formulation analogous to the dual-context typing judgement of the guarded lambda calculus. Compared to standard intuitionistic higher-order logic, the logic GHOL has two additional constructs, corresponding to additional constructs in the guarded lambda calculus. These are the later modality ($\triangleright$) *on propositions*, with delayed substitutions, which expresses that a proposition holds one time unit into the future, and the "always" modality $\square$, which expresses that a proposition holds at all times. Formulas are defined by the grammar:

$$\phi, \psi ::= \top \mid \phi \wedge \psi \mid \phi \vee \psi \mid \neg\psi \mid \forall x.\phi \mid \exists x.\phi \mid \triangleright [x_1 \leftarrow t_1 \ldots x_n \leftarrow t_n] .\phi \mid \square\phi$$

The basic judgement of the logic is $\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \phi$ where Σ is a logical context for Δ (that is, a list of formulas well-formed in Δ) and Ψ is another logical context for the dual context $\Delta \mid \Gamma$. The formulas in context Σ must be *constant* propositions. We say that a proposition ϕ is *constant* if it is well-typed in context $\Delta \mid \cdot$ and moreover if every occurrence of the later modality in ϕ is under the $\Box$ modality. Selected rules are displayed in Fig. 3. We highlight [Loeb] induction, which is the key to reasoning about fixpoints: to prove that ϕ holds now, one can assume that it holds in the future. The interpretation of the formula $\Delta \mid \Gamma \vdash \phi$ is a subobject of the interpretation $[\![\Delta \mid \Gamma]\!]$. Concretely the interpretation A of $\Delta \mid \Gamma \vdash \phi$ is a family $\{A_i\}_{i=0}^{\infty}$ of sets such that $A_i \subseteq [\![\Delta \mid \Gamma]\!]_i$. This family must satisfy the property that if $x \in A_{i+1}$ then $r_i(x) \in A_i$ where r_i are the restriction functions of $[\![\Delta \mid \Gamma]\!]$. The interpretation of formulas is defined by induction on the typing derivation. In the interpretation of the context $\Delta \mid \Sigma \mid \Gamma \mid \Psi$ the formulas in Σ are interpreted with the added $\Box$ modality. Moreover all formulas ϕ in Σ are typeable in the context $\Delta \mid \cdot \vdash \phi$ and thus their interpretations are subsets of $[\![\Box\Delta]\!]$. We treat these subsets of $[\![\Delta \mid \Gamma]\!]$ in the obvious way.

The cases for the semantics of the judgement $\Delta \mid \Gamma \vdash \phi$ can be found in the appendix. It can be shown that this logic is sound with respect to its model in the topos of trees.

Theorem 2 (Soundness of the semantics). *The semantics of guarded higher-order logic is sound: if $\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \phi$ is derivable then for all $n \in \mathbb{N}$, $[\![\Box\Sigma]\!]_n \cap [\![\Psi]\!]_n \subseteq [\![\phi]\!]$.*

In addition, Guarded HOL is expressive enough to axiomatize standard probabilities over discrete sets. This axiomatization can be used to define the $\diamond$ modality directly in Guarded HOL (as opposed to our relational proof system, were we use it as a primitive). Furthermore, we can derive from this axiomatization additional rules to reason about couplings, which can be seen in Fig. 4. These rules will be the key to proving the soundness of the probabilistic fragment of the relational proof system, and can be shown to be sound themselves.

Proposition 2 (Soundness of derived rules). *The additional rules are sound.*

6 Relational Proof System

We complete the formal description of the system by describing the proof rules for the non-probabilistic fragment of the relational proof system (the rules of the probabilistic fragment were described in Sect. 3.2).

6.1 Proof Rules

The rules for core λ-calculus constructs are identical to those of [2]; for convenience, we present a selection of the main rules in Fig. 7 in the appendix.

$$\frac{\phi \in \Psi}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \phi}\ \mathsf{AX_U} \qquad \frac{\phi \in \Sigma}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \phi}\ \mathsf{AX_G} \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash t' : \tau \quad t \equiv t'}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash t = t'}\ \mathsf{CONV}$$

$$\frac{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \phi[t/x] \quad \Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash t = u}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \phi[u/x]}\ \mathsf{SUBST} \qquad \frac{\Delta \mid \Sigma \mid \Gamma \mid \Psi, \triangleright\phi \vdash \phi}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \phi}\ \mathsf{Loeb}$$

$$\frac{\Delta \mid \Sigma \mid \Gamma, x_1 : A_1, \ldots, x_n : A_n \mid \Psi \vdash \phi \quad \Delta \mid \Gamma \vdash t_1 : \triangleright A_1 \quad \ldots \quad \Delta \mid \Gamma \vdash t_n : \triangleright A_n}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \triangleright[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n].\phi}\ \triangleright_{\mathsf{I}}$$

$$\frac{\Delta \mid \Sigma \mid \cdot \mid \cdot \vdash \triangleright[x_1 \leftarrow t_1 \ldots x_n \leftarrow t_n].\phi \quad \Delta \mid \bullet \vdash t_1 : \triangleright A_1 \quad \ldots \quad \Delta \mid \bullet \vdash t_n : \triangleright A_n}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \phi[\text{prev } t_1/x_1] \ldots [\text{prev } t_n/x_n]}\ \triangleright_{\mathsf{E}}$$

$$\frac{\begin{array}{c}\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \triangleright[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n].\psi \qquad \Delta \mid \Gamma \vdash t_1 : \triangleright A_1 \ \ldots \ \Delta \mid \Gamma \vdash t_n : \triangleright A_n \\ \Delta \mid \Sigma \mid \Gamma, x_1 : A_1, \ldots, x_n : A_n \mid \Psi, \psi \vdash \phi\end{array}}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \triangleright[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n].\phi}\ \triangleright_{\mathsf{App}}$$

$$\frac{\Delta \mid \Sigma \mid \cdot \mid \cdot \vdash \phi}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \Box\phi}\ \Box_{\mathsf{I}} \qquad \frac{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \Box\psi \quad \Delta \mid \Sigma, \psi \mid \Gamma \mid \Psi \vdash \phi}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \phi}\ \Box_{\mathsf{E}}$$

Fig. 3. Selected Guarded Higher-Order Logic rules

$$\frac{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \diamond_{[x_1 \leftarrow t_1, x_2 \leftarrow t_2]}\phi \qquad \Delta \mid \Sigma \mid \Gamma, x_1 : C_1, x_2 : C_2 \mid \Psi, \phi \vdash \psi}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \diamond_{[x_1 \leftarrow t_1, x_2 \leftarrow t_2]}\psi}\ \mathsf{MONO2}$$

$$\frac{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \phi[t_1/x_1][t_2/x_2]}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \diamond_{[x_1 \leftarrow \mathsf{munit}(t_1), x_2 \leftarrow \mathsf{munit}(t_2)]}\phi}\ \mathsf{UNIT2}$$

$$\frac{\begin{array}{c}\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \diamond_{[x_1 \leftarrow t_1, x_2 \leftarrow t_2]}\phi \\ \Delta \mid \Sigma \mid \Gamma, x_1 : C_1, x_2 : C_2 \mid \Psi, \phi \vdash \diamond_{[y_1 \leftarrow t'_1, y_2 \leftarrow t'_2]}\psi\end{array}}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \diamond_{[y_1 \leftarrow \mathsf{let}\ x_1 = t_1\ \mathsf{in}\ t'_1, y_2 \leftarrow \mathsf{let}\ x_2 = t_2\ \mathsf{in}\ t'_2]}\psi}\ \mathsf{MLET2}$$

$$\frac{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \diamond_{[x_1 \leftarrow t_1]}\phi \qquad \Delta \mid \Sigma \mid \Gamma, x_1 : C_1 \mid \Psi, \phi \vdash \diamond_{[y_1 \leftarrow t'_1, y_2 \leftarrow t'_2]}\psi}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \diamond_{[y_1 \leftarrow \mathsf{let}\ x_1 = t_1\ \mathsf{in}\ t'_1, y_2 \leftarrow t'_2]}\psi}\ \mathsf{MLET\text{-}L}$$

Fig. 4. Derived rules for probabilistic constructs

We briefly comment on the two-sided rules for the new constructs (Fig. 5). The notation Ω abbreviates a context $\Delta \mid \Sigma \mid \Gamma \mid \Psi$. The rule [Next] relates two terms that have a $\triangleright$ term constructor at the top level. We require that both have one term in the delayed substitutions and that they are related pairwise. Then this relation is used to prove another relation between the main terms. This rule can be generalized to terms with more than one term in the delayed substitution. The rule [Prev] proves a relation between terms from the same delayed relation by applying prev to both terms. The rule [Box] proves a relation between two boxed terms if the same relation can be proven in a constant context. Dually, [LetBox] uses a relation between two boxed terms to prove a relation between their unboxings. [LetConst] is similar to [LetBox], but it requires instead a relation between two constant terms, rather than explicitly $\Box$-ed terms. The rule [Fix] relates two fixpoints following the [Loeb] rule from Guarded HOL. Notice that in the premise, the fixpoints need to appear in the delayed substitution so that the inductive hypothesis is well-formed. The rule [Cons] proves relations on streams from relations between their heads and tails, while [Head] and [Tail] behave as converses of [Cons].

Figure 6 contains the one-sided versions of the rules. We only present the left-sided versions as the right-sided versions are completely symmetric. The rule [Next-L] relates at ϕ a term that has a $\triangleright$ with a term that does not have a $\triangleright$. First, a unary property ϕ' is proven on the term u in the delayed substitution, and it is then used as a premise to prove ϕ on the terms with delays removed. Rules for proving unary judgements can be found in the appendix. Similarly, [LetBox-L] proves a unary property on the term that gets unboxed and then uses it as a precondition. The rule [Fix-L] builds a fixpoint just on the left, and relates it with an arbitrary term t_2 at a property ϕ. Since ϕ may contain the variable $\mathbf{r}_2$ which is not in the context, it has to be replaced when adding $\triangleright\phi$ to the logical context in the premise of the rule. The remaining rules are similar to their two-sided counterparts.

6.2 Metatheory

We review some of the most interesting metatheoretical properties of our relational proof system, highlighting the equivalence with Guarded HOL.

Theorem 3 (Equivalence with Guarded HOL). *For all contexts Δ, Γ; types σ_1, σ_2; terms t_1, t_2; sets of assertions Σ, Ψ; and assertions ϕ:*

$$\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi \quad \Longleftrightarrow \quad \Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \phi[t_1/\mathbf{r}_1][t_2/\mathbf{r}_2]$$

The forward implication follows by induction on the given derivation. The reverse implication is immediate from the rule which allows to fall back on Guarded HOL in relational proofs. (Rule [SUB] in the appendix). The full proof is in the appendix. The consequence of this theorem is that the syntax-directed, relational proof system we have built on top of Guarded HOL does not lose expressiveness.

The intended semantics of a judgement $\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash t_1 : A_1 \sim t_2 : A_2 \mid \phi$ is that, for every valuation $\delta \models \Delta$, $\gamma \models \Gamma$, if $[\![\Sigma]\!](\delta)$ and $[\![\Psi]\!](\delta, \gamma)$, then

$$[\![\phi]\!](\delta, \gamma[\mathbf{r}_1 \leftarrow [\![t_1]\!](\delta, \gamma), \mathbf{r}_2 \leftarrow [\![t_2]\!](\delta, \gamma)])$$

Since Guarded HOL is sound with respect to its semantics in the topos of trees, and our relational proof system is equivalent to Guarded HOL, we obtain that our relational proof system is also sound in the topos of trees.

Corollary 2 (Soundness and consistency). *If $\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash t_1 : \sigma_2 \sim t_2 : \sigma_2 \mid \phi$, then for every valuation $\delta \models \Delta$, $\gamma \models \Gamma$:*

$$[\![\Delta \vdash \Box\Sigma]\!](\delta) \wedge [\![\Delta \mid \Gamma \vdash \Psi]\!](\delta, \gamma) \Rightarrow$$
$$[\![\Delta \mid \Gamma, \mathbf{r}_1 : \sigma_1, \mathbf{r}_1 : \sigma_2 \vdash \phi]\!](\delta, \gamma[\mathbf{r}_1 \leftarrow [\![\Delta \mid \Gamma \vdash t_1]\!](\delta, \gamma)][\mathbf{r}_2 \leftarrow [\![\Delta \mid \Gamma \vdash t_2]\!](\delta, \gamma)])$$

In particular, there is no proof of $\Delta \mid \emptyset \mid \Gamma \mid \emptyset \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \bot$.

$$\frac{\begin{array}{c}\Delta \mid \Sigma \mid \Gamma, x_1 : A_1, x_2 : A_2 \mid \Psi, \phi'[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \vdash t_1 : A_1 \sim t_2 : A_2 \mid \phi \\ \Omega \vdash u_1 : \triangleright A_1 \sim u_2 : \triangleright A_2 \mid \triangleright[\mathbf{r}_1, \mathbf{r}_2 \leftarrow \mathbf{r}_1, \mathbf{r}_2].\phi'\end{array}}{\Omega \vdash \triangleright[x_1 \leftarrow u_1].t_1 : \triangleright A_1 \sim \triangleright[x_2 \leftarrow u_2].t_2 : \triangleright A_2 \mid \triangleright[x_1 \leftarrow u_1, x_2 \leftarrow u_2, \mathbf{r}_1 \leftarrow \mathbf{r}_1, \mathbf{r}_2 \leftarrow \mathbf{r}_2].\phi} \textsf{Next}$$

$$\frac{\Delta \mid \Sigma \mid \cdot \mid \cdot \vdash t_1 : \triangleright A_1 \sim t_2 : \triangleright A_2 \mid \triangleright[\mathbf{r}_1, \mathbf{r}_2 \leftarrow \mathbf{r}_1, \mathbf{r}_2].\phi}{\Omega \vdash \text{prev } t_1 : A_1 \sim \text{prev } t_2 : A_2 \mid \phi} \textsf{Prev}$$

$$\frac{\Delta \mid \Sigma \mid \cdot \mid \cdot \vdash t_1 : A_1 \sim t_2 : A_2 \mid \phi}{\Omega \vdash \text{box } t_1 : \Box A_1 \sim \text{box } t_2 : \Box A_2 \mid \Box\phi[\text{letb } x_1 \leftarrow \mathbf{r}_1 \text{ in } x_1/\mathbf{r}_1][\text{letb } x_2 \leftarrow \mathbf{r}_2 \text{ in } x_2/\mathbf{r}_2]} \textsf{Box}$$

$$\frac{\begin{array}{c}\Omega \vdash u_1 : \Box B_1 \sim u_2 : \Box B_2 \mid \Box\phi[\text{letb } x_1 \leftarrow \mathbf{r}_1 \text{ in } x_1/\mathbf{r}_1][\text{letb } x_2 \leftarrow \mathbf{r}_2 \text{ in } x_2/\mathbf{r}_2] \\ \Delta, x_1 : B_1, x_2 : B_2 \mid \Sigma, \phi[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \mid \Gamma \mid \Psi \vdash t_1 : A_1 \sim t_2 : A_2 \mid \phi'\end{array}}{\Omega \vdash \text{letb } x_1 \leftarrow u_1 \text{ in } t_1 : A_1 \sim \text{letb } x_2 \leftarrow u_2 \text{ in } t_2 : A_2 \mid \phi'} \textsf{LetBox}$$

$$\frac{\begin{array}{c}B_1, B_2, \phi \text{ constant} \qquad FV(\phi) \cap FV(\Gamma) = \emptyset \qquad \Omega \vdash u_1 : B_1 \sim u_2 : B_2 \mid \phi \\ \Delta, x_1 : B_1, x_2 : B_2 \mid \Sigma, \phi[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \mid \Gamma \mid \Psi \vdash t_1 : A_1 \sim t_2 : A_2 \mid \phi'\end{array}}{\Omega \vdash \text{letc } x_1 \leftarrow u_1 \text{ in } t_1 : A_1 \sim \text{letc } x_2 \leftarrow u_2 \text{ in } t_2 : A_2 \mid \phi'} \textsf{LetConst}$$

$$\frac{\Delta \mid \Sigma \mid \Gamma, f_1 : \triangleright A_1, f_2 : \triangleright A_2 \mid \Psi, \triangleright[\mathbf{r}_1, \mathbf{r}_2 \leftarrow f_1, f_2].\phi \vdash t_1 : A_1 \sim t_2 : A_2 \mid \phi}{\Omega \vdash \text{fix } f_1.\ t_1 : A_1 \sim \text{fix } f_2.\ t_2 : A_2 \mid \phi} \textsf{Fix}$$

$$\frac{\begin{array}{c}\Omega \vdash x_1 : A_1 \sim x_2 : A_2 \mid \phi_h \qquad \Omega \vdash xs_1 : \triangleright \text{Str}_{A_1} \sim xs_2 : \triangleright \text{Str}_{A_2} \mid \phi_t \\ \Omega \vdash \forall x_1, x_2, s_1, s_2.\phi_h[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \Rightarrow \phi_t[s_1/\mathbf{r}_1][s_2/\mathbf{r}_2] \Rightarrow \phi[x_1 :: s_1/\mathbf{r}_1][x_2 :: s_2/\mathbf{r}_2]\end{array}}{\Omega \vdash x_1 :: s_1 : \text{Str}_{A_1} \sim x_2 :: s_2 : \text{Str}_{A_2} \mid \phi} \textsf{Cons}$$

$$\frac{\Omega \vdash t_1 : \text{Str}_{A_1} \sim t_1 : \text{Str}_{A_1} \mid \phi[hd\ \mathbf{r}_1/\mathbf{r}_1][hd\ \mathbf{r}_2/\mathbf{r}_2]}{\Omega \vdash hd\ t_1 : A_1 \sim hd\ t_2 : A_2 \mid \phi} \textsf{Head}$$

$$\frac{\Omega \vdash t_1 : \text{Str}_{A_1} \sim t_2 : \text{Str}_{A_2} \mid \phi[tl\ \mathbf{r}_1/\mathbf{r}_1][tl\ \mathbf{r}_2/\mathbf{r}_2]}{\Omega \vdash tl\ t_1 : \triangleright \text{Str}_{A_1} \sim tl\ t_2 : \triangleright \text{Str}_{A_2} \mid \phi} \textsf{Tail}$$

Fig. 5. Two-sided rules for Guarded RHOL

6.3 Shift Couplings Revisited

We give further details on how to prove the example with shift couplings from Sect. 3.3. (Additional examples of relational reasoning on non-probabilistic streams can be found in the appendix) Recall the step functions:

$$\begin{aligned} \text{step} &\triangleq \lambda x.\textsf{let } z = \mathcal{U}_{\{-1,1\}} \textsf{ in munit}(z + x) \\ \text{lstep2} &\triangleq \lambda x.\textsf{let } z = \mathcal{U}_{\{-1,1\}} \textsf{ in let } b = \mathcal{U}_{\{0,1\}} \textsf{ in munit}(x + 2 * z * b) \end{aligned}$$

We axiomatize the predicate $\text{All}_{2,1}$, which relates the element at position $2i$ in one stream to the element at position i in another stream, as follows.

$$\begin{array}{l} \forall x_1 x_2 xs_1 xs_2 y_1.\phi[z_1/x_1][z_2/x_2] \Rightarrow \\ \quad \triangleright[ys_1 \leftarrow xs_1] . \triangleright[zs_1 \leftarrow ys_1, ys_2 \leftarrow xs_2] . \text{All}_{2,1}(zs_1, ys_2, z_1.z_2.\phi) \Rightarrow \\ \quad\quad \text{All}_{2,1}(x_1{::}y_1{::}xs_1, x_2{::}xs_2, z_1.z_2.\phi) \end{array}$$

In fact, we can assume that, in general, we have a family of All_{m_1,m_2} predicates relating two streams at positions $m_1 \cdot i$ and $m_2 \cdot i$ for every i.

$$\frac{\begin{array}{c}\Delta \mid \Sigma \mid \Gamma, x_1 : B_1 \mid \Psi, \phi'[x_1/\mathbf{r}] \vdash t_1 : A_1 \sim t_2 : A_2 \mid \phi \\ \Omega \vdash u_1 : \triangleright B_1 \mid \triangleright[\mathbf{r} \leftarrow \mathbf{r}].\phi'\end{array}}{\Omega \vdash \triangleright[x_1 \leftarrow u_1].t_1 : \triangleright A_1 \sim t_2 : A_2 \mid \triangleright[x_1 \leftarrow u_1, \mathbf{r}_1 \leftarrow \mathbf{r}_1].\phi}\ \mathsf{Next\text{-}L}$$

$$\frac{\Delta \mid \Sigma \mid \cdot \mid \cdot \vdash t_1 : \triangleright A_1 \sim t_2 : A_2 \mid \triangleright[\mathbf{r}_1 \leftarrow \mathbf{r}_1].\phi}{\Delta \mid \Sigma \mid \Gamma_1; \Gamma_2 \mid \Psi_1; \Psi_2 \vdash \text{prev } t_1 : A_1 \sim t_2 : A_2 \mid \phi}\ \mathsf{Prev\text{-}L}$$

$$\frac{\begin{array}{c}\Delta \mid \Sigma \mid \Gamma_2 \mid \Psi_2 \vdash t_1 : A_1 \sim t_2 : A_2 \mid \phi \\ FV(t_1) \not\subseteq FV(\Gamma_2) \qquad FV(\Psi_2) \subseteq FV(\Gamma_2)\end{array}}{\Delta \mid \Sigma \mid \Gamma_1; \Gamma_2 \mid \Psi_1; \Psi_2 \vdash \text{box } t_1 : \Box A_1 \sim t_2 : A_2 \mid \Box\phi[\text{letb } x_1 \leftarrow \mathbf{r}_1 \text{ in } x_1/\mathbf{r}_1]}\ \mathsf{Box\text{-}L}$$

$$\frac{\begin{array}{c}\Omega \vdash u_1 : \Box B_1 \mid \Box\phi[\text{letb } x_1 \leftarrow \mathbf{r}_1 \text{ in } x_1/\mathbf{r}] \\ \Delta, x_1 : B_1 \mid \Sigma, \phi[x_1/\mathbf{r}] \mid \Gamma \mid \Psi \vdash t_1 : A_1 \sim t_2 : A_2 \mid \phi'\end{array}}{\Omega \vdash \text{letb } x_1 \leftarrow u_1 \text{ in } t_1 : A_1 \sim t_2 : A_2 \mid \phi'}\ \mathsf{LetBox\text{-}L}$$

$$\frac{\begin{array}{c}B_1, \phi \text{ constant} \qquad FV(\phi) \cap FV(\Gamma) = \emptyset \qquad \Omega \vdash u_1 : B_1 \mid \phi \\ \Delta, x_1 : B_1 \mid \Sigma, \phi[x_1/\mathbf{r}] \mid \Gamma \mid \Psi \vdash t_1 : A_1 \sim t_2 : A_2 \mid \phi'\end{array}}{\Omega \vdash \text{letc } x_1 \leftarrow u_1 \text{ in } t_1 : A_1 \sim t_2 : A_2 \mid \phi'}\ \mathsf{LetConst\text{-}L}$$

$$\frac{\Delta \mid \Sigma \mid \Gamma, f_1 : \triangleright A_1 \mid \Psi, \triangleright[\mathbf{r}_1 \leftarrow f_1].(\phi[t_2/\mathbf{r}_2]) \vdash t_1 : A_1 \sim t_2 : A_2 \mid \phi}{\Delta \mid \Sigma \mid \Gamma \mid \Psi \vdash \text{fix } f_1.\ t_1 : A_1 \sim t_2 : A_2 \mid \phi}\ \mathsf{Fix\text{-}L}$$

$$\frac{\begin{array}{c}\Omega \vdash x_1 : A_1 \sim t_2 : A_2 \mid \phi_h \qquad \Omega \vdash xs_1 : \triangleright \text{Str}_{A_1} \sim t_2 : A_2 \mid \phi_t \\ \Omega \vdash \forall x_1, x_2, xs_1.\phi_h[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \Rightarrow \phi_t[xs_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \Rightarrow \phi[x_1 :: xs_1/\mathbf{r}_1][x_2/\mathbf{r}_2]\end{array}}{\Omega \vdash x_1 :: xs_1 : \text{Str}_{A_1} \sim t_2 : A_2 \mid \phi}\ \mathsf{Cons\text{-}L}$$

$$\frac{\Omega \vdash t_1 : \text{Str}_{A_1} \sim t_1 : A_2 \mid \phi[hd\ \mathbf{r}_1/\mathbf{r}_1]}{\Omega \vdash hd\ t_1 : A_1 \sim t_2 : A_2 \mid \phi}\ \mathsf{Head\text{-}L}$$

$$\frac{\Omega \vdash t_1 : \text{Str}_{A_1} \sim t_2 : A_2 \mid \phi[tl\ \mathbf{r}_1/\mathbf{r}_1]}{\Omega \vdash tl\ t_1 : \triangleright \text{Str}_{A_1} \sim t_2 : A_2 \mid \phi}\ \mathsf{Tail\text{-}L}$$

Fig. 6. One-sided rules for Guarded RHOL

We can now express the existence of a shift coupling by the statement:

$$p_1 = p_2 \vdash \mathsf{markov}(p_1, \text{step}) \sim \mathsf{markov}(p_2, \text{lstep2}) \mid \diamond_{\left[\substack{y_1 \leftarrow \mathbf{r}_1 \\ y_2 \leftarrow \mathbf{r}_2}\right]} \text{All}_{2,1}(y_1, y_2, z_1.z_2.z_1 = z_2)$$

For the proof, we need to introduce an asynchronous rule for Markov chains:

$$\frac{\begin{array}{c}\Omega \vdash t_1 : C_1 \sim t_2 : C_2 \mid \phi \\ \Omega \vdash (\lambda x_1.\mathsf{let}\ x_1' = h_1\ x_1\ \mathsf{in}\ h_1\ x_1') : C_1 \to \mathsf{D}(C_1) \sim h_2 : C_2 \to \mathsf{D}(C_2) \mid \\ \forall x_1 x_2.\phi[x_1/z_1][x_2/z_2] \Rightarrow \diamond_{[z_1 \leftarrow \mathbf{r}_1\ x_1, z_2 \leftarrow \mathbf{r}_2\ x_2]}\phi\end{array}}{\begin{array}{c}\Omega \vdash \mathsf{markov}(t_1, h_1) : \mathsf{D}(\text{Str}_{C_1}) \sim \mathsf{markov}(t_2, h_2) : \mathsf{D}(\text{Str}_{C_2}) \mid \\ \diamond_{[y_1 \leftarrow \mathbf{r}_1, y_2 \leftarrow \mathbf{r}_2]} \text{All}_{2,1}(y_1, y_2, z_1.z_2.\phi)\end{array}}\ \mathsf{Markov\text{-}2\text{-}1}$$

This asynchronous rule for Markov chains shares the motivations of the rule for loops proposed in [6]. Note that one can define a rule [Markov-m-n] for arbitrary m and n to prove a judgement of the form $\text{All}_{m,n}$ on two Markov chains.

We show the proof of the shift coupling. By equational reasoning, we get:

$$\begin{aligned}&\lambda x_1.\mathsf{let}\ x_1' = h_1\ x_1\ \mathsf{in}\ h_1\ x_1' \ \equiv\ \lambda x_1.\mathsf{let}\ z_1 = \mathcal{U}_{\{-1,1\}}\ \mathsf{in}\ h_1\ (z_1 + x_1)\\ &\equiv\ \lambda x_1.\mathsf{let}\ z_1 = \mathcal{U}_{\{-1,1\}}\ \mathsf{in}\ \mathsf{let}\ z_1' = \mathcal{U}_{\{-1,1\}}\ \mathsf{in}\ \mathsf{munit}(z_1' + z_1 + x_1')\end{aligned}$$

and the only interesting premise of [Markov-2-1] is:

$$\left.\begin{array}{lcl}\lambda x_1.\ \mathsf{let}\ z_1 = \mathcal{U}_{\{-1,1\}}\ \mathsf{in} & & \lambda x_2.\ \mathsf{let}\ z_2 = \mathcal{U}_{\{-1,1\}}\ \mathsf{in}\\ \quad\ \ \mathsf{let}\ z_1' = \mathcal{U}_{\{-1,1\}}\ \mathsf{in} & \sim & \quad\ \ \mathsf{let}\ b_2 = \mathcal{U}_{\{1,0\}}\ \mathsf{in}\\ \quad\ \ \mathsf{munit}(z_1' + z_1 + x_1') & & \quad\ \ \mathsf{munit}(x_2 + 2 * b_2 * z_2)\end{array}\ \right|\ \begin{array}{c}\forall x_1 x_2. x_1 = x_2 \Rightarrow\\ \mathbf{r}_1\ x_1 \overset{\diamond}{=} \mathbf{r}_2\ x_2\end{array}$$

Couplings between z_1 and z_2 and between z_1' and b_2 can be found by simple computations. This completes the proof.

7 Related Work

Our probabilistic guarded λ-calculus and the associated logic Guarded HOL build on top of the guarded λ-calculus and its internal logic [1]. The guarded λ-calculus has been extended to guarded dependent type theory [13], which can be understood as a theory of guarded refinement types and as a foundation for proof assistants based on guarded type theory. These systems do not reason about probabilities, and do not support syntax-directed (relational) reasoning, both of which we support.

Relational models for higher-order programming languages are often defined using logical relations. [16] showed how to use second-order logic to define and reason about logical relations for the second-order lambda calculus. Recent work has extended this approach to logical relations for higher-order programming languages with computational effects such as nontermination, general references, and concurrency [17–20]. The logics used in *loc. cit.* are related to our work in two ways: (1) the logics in *loc. cit.* make use of the later modality for reasoning about recursion, and (2) the models of the logics in *loc. cit.* can in fact be defined using guarded type theory. Our work is more closely related to Relational Higher Order Logic [2], which applies the idea of logic-enriched type theories [21,22] to a relational setting. There exist alternative approaches for reasoning about relational properties of higher-order programs; for instance, [23] have recently proposed to use monadic reification for reducing relational verification of F^* to proof obligations in higher-order logic.

A series of work develops reasoning methods for probabilistic higher-order programs for different variations of the lambda calculus. One line of work has focused on operationally-based techniques for reasoning about contextual equivalence of programs. The methods are based on probabilistic bisimulations [24,25] or on logical relations [26]. Most of these approaches have been developed for languages with discrete distributions, but recently there has also been work on languages with continuous distributions [27,28]. Another line of work has focused on denotational models, starting with the seminal work in [29]. Recent work includes support for relational reasoning about equivalence of programs with continuous distributions for a total programming language [30]. Our approach is most closely related to prior work based on relational refinement types for higher-order probabilistic programs. These were initially considered by [31] for a stateful fragment of F^*, and later by [32,33] for a pure language. Both systems are specialized to building probabilistic couplings; however, the latter support approximate probabilistic couplings, which yield a natural interpretation of differential privacy [34], both in its vanilla and approximate forms (i.e. ϵ- and (ϵ, δ)-privacy). Technically, approximate couplings are modelled as a graded monad, where the index of the monad tracks the privacy budget (ϵ or (ϵ, δ)). Both systems are strictly syntax-directed, and cannot reason about computations that have different types or syntactic structures, while our system can.

8 Conclusion

We have developed a probabilistic extension of the (simply typed) guarded λ-calculus, and proposed a syntax-directed proof system for relational verification. Moreover, we have verified a series of examples that are beyond the reach of prior work. Finally, we have proved the soundness of the proof system with respect to the topos of trees.

There are several natural directions for future work. One first direction is to enhance the expressiveness of the underlying simply typed language. For instance, it would be interesting to introduce clock variables and some type dependency as in [13], and extend the proof system accordingly. This would allow us, for example, to type the function taking the n-th element of a *guarded* stream, which cannot be done in the current system. Another exciting direction is to consider approximate couplings, as in [32,33], and to develop differential privacy for infinite streams—preliminary work in this direction, such as [35], considers very large lists, but not arbitrary streams. A final direction would be to extend our approach to continuous distributions to support other application domains.

Acknowledgments. We would like to thank the anonymous reviewers for their time and their helpful input. This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU), by a research grant (12386, Guarded Homotopy Type Theory) from the VILLUM foundation, and by NSF under grant 1718220.

References

1. Clouston, R., Bizjak, A., Grathwohl, H.B., Birkedal, L.: The guarded lambda-calculus: programming and reasoning with guarded recursion for coinductive types. Log. Methods Comput. Sci. **12**(3) (2016)
2. Aguirre, A., Barthe, G., Gaboardi, M., Garg, D., Strub, P.: A relational logic for higher-order programs. PACMPL **1**(ICFP), 21:1–21:29 (2017)
3. Lindvall, T.: Lectures on the Coupling Method. Courier Corporation (2002)

4. Thorisson, H.: Coupling, Stationarity, and Regeneration. Springer, New York (2000)
5. Barthe, G., Espitau, T., Grégoire, B., Hsu, J., Stefanesco, L., Strub, P.-Y.: Relational reasoning via probabilistic coupling. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 387–401. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_27
6. Barthe, G., Grégoire, B., Hsu, J., Strub, P.: Coupling proofs are probabilistic product programs. In: POPL 2017, Paris, France, 18–20 January 2017 (2017)
7. Strassen, V.: The existence of probability measures with given marginals. Ann. Math. Stat. **36**, 423–439 (1965)
8. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy, pp. 11–20 (1982)
9. Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multi-party computation for data mining applications. Int. J. Inf. Sec. **11**(6), 403–418 (2012)
10. Cramer, R., Damgard, I.B., Nielsen, J.B.: Secure Multiparty Computation and Secret Sharing, 1st edn. Cambridge University Press, New York (2015)
11. Barthe, G., Espitau, T., Grégoire, B., Hsu, J., Strub, P.: Proving uniformity and independence by self-composition and coupling. CoRR abs/1701.06477 (2017)
12. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: Proceedings of Foundations of Computer Science. IEEE (2001)
13. Bizjak, A., Grathwohl, H.B., Clouston, R., Møgelberg, R.E., Birkedal, L.: Guarded dependent type theory with coinductive types. In: Jacobs, B., Löding, C. (eds.) FoSSaCS 2016. LNCS, vol. 9634, pp. 20–35. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49630-5_2
14. McBride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. **18**(1), 1–13 (2008)
15. Birkedal, L., Møgelberg, R.E., Schwinghammer, J., Støvring, K.: First steps in synthetic guarded domain theory: step-indexing in the topos of trees. Log. Methods Comput. Sci. **8**(4) (2012)
16. Plotkin, G., Abadi, M.: A logic for parametric polymorphism. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 361–375. Springer, Heidelberg (1993). https://doi.org/10.1007/BFb0037118
17. Dreyer, D., Ahmed, A., Birkedal, L.: Logical step-indexed logical relations. Log. Methods Comput. Sci. **7**(2) (2011)
18. Turon, A., Dreyer, D., Birkedal, L.: Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In: Morrisett, G., Uustalu, T. (eds.) ICFP 2013, Boston, MA, USA, 25–27 September 2013. ACM (2013)
19. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: Castagna, G., Gordon, A.D. (eds.) POPL 2017, Paris, France, 18–20 January 2017. ACM (2017)
20. Krogh-Jespersen, M., Svendsen, K., Birkedal, L.: A relational model of types-and-effects in higher-order concurrent separation logic. In: POPL 2017, Paris, France, 18–20 January 2017, pp. 218–231 (2017)
21. Aczel, P., Gambino, N.: Collection principles in dependent type theory. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 1–23. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45842-5_1
22. Aczel, P., Gambino, N.: The generalised type-theoretic interpretation of constructive set theory. J. Symb. Log. **71**(1), 67–103 (2006)

23. Grimm, N., Maillard, K., Fournet, C., Hritcu, C., Maffei, M., Protzenko, J., Rastogi, A., Swamy, N., Béguelin, S.Z.: A monadic framework for relational verification (functional pearl). CoRR abs/1703.00055 (2017)
24. Crubillé, R., Dal Lago, U.: On probabilistic applicative bisimulation and call-by-value λ-calculi. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 209–228. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_12
25. Sangiorgi, D., Vignudelli, V.: Environmental bisimulations for probabilistic higher-order languages. In: Bodík, R., Majumdar, R. (eds.) POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016. ACM (2016)
26. Bizjak, A., Birkedal, L.: Step-indexed logical relations for probability. In: Pitts, A. (ed.) FoSSaCS 2015. LNCS, vol. 9034, pp. 279–294. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46678-0_18
27. Borgström, J., Lago, U.D., Gordon, A.D., Szymczak, M.: A lambda-calculus foundation for universal probabilistic programming. In: Garrigue, J., Keller, G., Sumii, E. (eds.) ICFP 2016, Nara, Japan, 18–22 September 2016. ACM (2016)
28. Culpepper, R., Cobb, A.: Contextual equivalence for probabilistic programs with continuous random variables and scoring. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 368–392. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_14
29. Jones, C., Plotkin, G.D.: A probabilistic powerdomain of evaluations. In: LICS 1989, Pacific Grove, California, USA, 5–8 June 1989. IEEE Computer Society (1989)
30. Staton, S., Yang, H., Wood, F., Heunen, C., Kammar, O.: Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In: LICS 2016, New York, NY, USA, 5–8 July 2016. ACM (2016)
31. Barthe, G., Fournet, C., Grégoire, B., Strub, P., Swamy, N., Béguelin, S.Z.: Probabilistic relational verification for cryptographic implementations. In: Jagannathan, S., Sewell, P. (eds.) POPL 2014 (2014)
32. Barthe, G., Gaboardi, M., Gallego Arias, E.J., Hsu, J., Roth, A., Strub, P.Y.: Higher-order approximate relational refinement types for mechanism design and differential privacy. In: POPL 2015, Mumbai, India, 15–17 January 2015 (2015)
33. Barthe, G., Farina, G.P., Gaboardi, M., Arias, E.J.G., Gordon, A., Hsu, J., Strub, P.: Differentially private Bayesian programming. In: CCS 2016, Vienna, Austria, 24–28 October 2016. ACM (2016)
34. Dwork, C., Roth, A.: The algorithmic foundations of differential privacy. Found. Trends Theor. Comput. Sci. **9**(3–4), 211–407 (2014)
35. Kellaris, G., Papadopoulos, S., Xiao, X., Papadias, D.: Differentially private event sequences over infinite streams. PVLDB **7**(12), 1155–1166 (2014)