Types and Effects

Failure is Not an Option
An Exceptional Type Theory

Pierre-Marie Pédrot[1(✉)] and Nicolas Tabareau[2]

[1] MPI-SWS, Saarbrücken, Germany
ppedrot@mpi-sws.org
[2] Inria, Nantes, France
nicolas.tabareau@inria.fr

Abstract. We define the *exceptional translation*, a syntactic translation of the Calculus of Inductive Constructions (CIC) into itself, that covers full dependent elimination. The new resulting type theory features call-by-name exceptions with decidable type-checking and canonicity, but at the price of inconsistency. Then, noticing parametricity amounts to Kreisel's realizability in this setting, we provide an additional layer on top of the exceptional translation in order to tame exceptions and ensure that all exceptions used locally are caught, leading to the *parametric exceptional translation* which fully preserves consistency. This way, we can consistently extend the logical expressivity of CIC with independence of premises, Markov's rule, and the negation of function extensionality while retaining η-expansion. As a byproduct, we also show that Markov's principle is not provable in CIC. Both translations have been implemented in a Coq plugin, which we use to formalize the examples.

1 Introduction

Monadic translations constitute a canonical way to add effects to pure functional languages [1]. Until recently, this technique was not available for type theories such as CIC because of complex interactions with dependency. In a recent paper [2], we have presented a generic way to extend the monadic translation to dependent types, using the *weaning translation*, as soon as the monad under consideration satisfies a crucial property: being self-algebraic. Indeed, in the same way that the universe of types $\square_i$ is itself a type (of a higher universe) in type theory, the type of algebras of a monad T

$$\Sigma A : \square_i .\, \mathrm{T}\ A \rightarrow A$$

needs to be itself an algebra of the monad to allow a correct translation of the universe. However, in general, the weaning translation does not interpret all of CIC because dependent elimination needs to be restricted to linear predicates, that is, those that are intuitively call-by-value [3]. In this paper, we study the particular case of the error monad, and show that its weaning translation can be simplified and tweaked so that full dependent elimination is valid.

© The Author(s) 2018
A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 245–271, 2018.
https://doi.org/10.1007/978-3-319-89884-1_9

This *exceptional translation* gives rise to a novel extension of CIC with new computational behaviours, namely call-by-name exceptions.[1] That is, the type theory induced by the exceptional translation features new operations to raise and catch exceptions. This new logical expressivity comes at a cost, as the resulting theory is not consistent anymore, although still being computationally relevant. This means that it is possible to prove a contradiction, but, thanks to a weak form of canonicity, only because of an unhandled exception. Furthermore, the translation allows us to reason directly in CIC on terms of the exceptional theory, letting us prove, e.g., that assuming some properties on its input, an exceptional function actually never raises an exception. We thus have a sound logical framework to prove safety properties about impure dependently-typed programs.

We then push this technique further by noticing that parametricity provides a systematic way to describe that a term is not allowed to produce uncaught exceptions, bridging the gap between Kreisel's modified realizability [4] and parametricity inside type theory [5]. This *parametric exceptional translation* ensures that no exception reaches toplevel, thus ensuring consistency of the resulting theory. Pure terms are automatically handled, while it is necessary to show parametricity manually for terms internally using exceptions. We exploit this computational extension of CIC to show various logical results over CIC.

Contributions

- We describe the *exceptional translation*, the first monadic translation for the error monad for CIC, including strong elimination of inductive types, resulting in a sound logical framework to reason about impure dependently-typed programs.
- We use parametricity to extend the exceptional translation, getting a consistent variant dubbed the *parametric exceptional translation*.
- We show that Markov's rule is admissible in CIC.
- We show that definitional η-expansion together with the negation of function extensionality is admissible in CIC.
- We show that there exists a syntactical model of CIC that validates the independence of premises (which is known to be generally not valid in intuitionistic logic [6]) and use it to recover the recent result of Coquand and Mannaa [7], *i.e.*, that Markov's principle is not provable in CIC.
- We provide a Coq plugin[2] that implements both translations and with which we have formalized all the examples.

Plan of the Paper. In Sect. 2, we describe the exceptional translation and the resulting new computational principles arising from it. In Sect. 3, we present the parametric variant of the exceptional translation. Section 4 is devoted to the

[1] The fact that the resulting exception are call-by-name is explained in detailed in [2] using a call-by-push-value decomposition. Intuitively, it comes from the fact that CIC is naturally call-by-name.

[2] The plugin is available at https://github.com/CoqHott/exceptional-tt.

$$A, B, M, N ::= \Box_i \mid x \mid M\ N \mid \lambda x : A.\, M \mid \Pi x : A.\, B$$

$$\Gamma, \Delta ::= \cdot \mid \Gamma, x : A$$

$$\frac{\vdash \Gamma \qquad i < j}{\Gamma \vdash \Box_i : \Box_j} \qquad\qquad \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \Box_i}{\Gamma, x : A \vdash M : B}$$

$$\frac{\Gamma \vdash A : \Box_i \qquad \Gamma, x : A \vdash B : \Box_j}{\Gamma \vdash \Pi x : A.\, B : \Box_{\max(i,j)}} \qquad \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \Box_i \qquad A \equiv B}{\Gamma \vdash M : A}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.\, B : \Box_i}{\Gamma \vdash \lambda x : A.\, M : \Pi x : A.\, B} \qquad \frac{\Gamma \vdash M : \Pi x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B\{x := N\}}$$

$$\frac{}{\vdash \cdot} \qquad \frac{\Gamma \vdash A : \Box_i}{\vdash \Gamma, x : A} \qquad\qquad \frac{\Gamma \vdash A : \Box_i}{\Gamma, x : A \vdash x : A}$$

$$(\lambda x : A.\, M)\ N \equiv M\{x := N\} \qquad\qquad \text{(congruence rules ommitted)}$$

Fig. 1. Typing rules of CC_ω

various logical results resulting from the parametric exceptional translations. In Sect. 5, we discuss possible extensions of the translation with negative records and an impredicative universe. Section 6 describes the Coq plugin and illustrates its use on a concrete example. We discuss related work in Sect. 7 and conclude in Sect. 8.

2 The Exceptional Translation

We define in this section the exceptional translation as a syntactic translation between type theories. We call the target theory $\mathcal{T}$, upon which we will make various assumptions depending on the objects we want to translate.

2.1 Adding Exceptions to CC_ω

In this section, we describe the exceptional translation over a purely negative theory, *i.e.*, featuring only universes and dependent functions, called CC_ω, which is presented in Fig. 1. This theory is a predicative version of the Calculus of Constructions [8], with an infinite hierarchy of universes $\Box_i$ instead of one impredicative sort. We assume from now on that $\mathcal{T}$ contains at least CC_ω itself.

The exceptional translation is a simplification of the weaning translation [2] applied to the error monad. Owing to the fact that it is specifically tailored for exceptions, this allows to give a more compact presentation of it.

Let $\mathbb{E} : \Box_0$ be a fixed type of exceptions in $\mathcal{T}$. The weaning translation for the error monad amounts to interpret types as algebras, *i.e.*, as inhabitants of the dependent sum $\Sigma A : \square_i .\, (A + \mathbb{E}) \rightarrow A$. In this paper, we take advantage of the fact that the algebra morphism restricted to A is always the identity. Thus every type just comes with a way to interpret failure on this type, i.e. types are intuitively interpreted as a pair of an $A : \square_i$ with a default (raise) function $A_\varnothing : \mathbb{E} \rightarrow A$. In practice, it is slightly more complicated as the universe of types itself is a type, so its interpretation must comes with a default function. We overcome this issue by assuming a term $\texttt{type}_i$, representing types that can raise exceptions. This type comes with two constructors: $\texttt{TypeVal}_i$ which allows to construct a $\texttt{type}_i$ from a type and a default function on this type ; and another constructor $\texttt{TypeErr}_i$ that represents the default function at the level of $\texttt{type}_i$. Furthermore, $\texttt{type}_i$ is equipped with an eliminator $\texttt{type\_elim}_i$ and thus can be thought of as an inductive definition. For simplicity, we axiomatize it instead of requiring inductive types in the target of the translation.

Definition 1. *We assume that $\mathcal{T}$ features the data below, where i, j indices stand for universe polymorphism.*

- $\Omega_i : \mathbb{E} \rightarrow \square_i$
- $\omega_i : \Pi e : \mathbb{E} .\, \Omega_i\ e$
- $\texttt{type}_i : \square_j$, *where* $i < j$
- $\texttt{TypeVal}_i : \Pi A : \square_i .\, (\mathbb{E} \rightarrow A) \rightarrow \texttt{type}_i$
- $\texttt{TypeErr}_i : \mathbb{E} \rightarrow \texttt{type}_i$
- $\texttt{type\_elim}_{i,j} :\ \Pi P : \texttt{type}_i \rightarrow \square_j .$ $(\Pi (A : \square_i)\, (A_\varnothing : \mathbb{E} \rightarrow A).\, P\ (\texttt{TypeVal}_i\ A\ A_\varnothing)) \rightarrow$ $(\Pi e : \mathbb{E} .\, P\ (\texttt{TypeErr}_i\ e)) \rightarrow \Pi T : \texttt{type}_i .\, P\ T$

subject to the following definitional equations:

$$\texttt{type\_elim}_{i,j}\ P\ p_v\ p_\varnothing\ (\texttt{TypeVal}_i\ A\ A_\varnothing) \equiv p_v\ A\ A_\varnothing$$

$$\texttt{type\_elim}_{i,j}\ P\ p_v\ p_\varnothing\ (\texttt{TypeErr}_i\ e) \equiv p_\varnothing\ e$$

The Ω term describes what it means for a type to fail, i.e. it ascribes a meaning to sequents of the form $\Gamma \vdash M : \texttt{fail}\ e$. In practice, it is irrelevant and can be chosen to be degenerate, e.g. $\Omega := \lambda \_ : \mathbb{E} .\, \texttt{unit}$.

In what follows, we often leave the universe indices implicit although they can be retrieved at the cost of more explicit annotations.

Before defining the exceptional translation we need to derive a term El[3] that recovers the underlying type from an inhabitant of type and Err that lifts the default function to this underlying type.

Definition 2. *From the data of Definition 1, we derive the following terms.*

$$\begin{array}{lll} \texttt{El}_i & : & \texttt{type}_i \rightarrow \square_i \\ & := & \lambda A : \texttt{type}_i .\, \texttt{type\_elim}\ (\lambda T : \texttt{type}_i .\, \square_i) \\ & & \quad (\lambda (A_0 : \square_i)\, (A_\varnothing : \mathbb{E} \rightarrow A_0).\, A_0)\ \Omega\ A \\ \texttt{Err}_i & : & \Pi A : \texttt{type}_i .\, \mathbb{E} \rightarrow \texttt{El}_i\ A \\ & := & \lambda (A : \texttt{type}_i)\, (e : \mathbb{E}).\, \texttt{type\_elim}\ \texttt{El}_i \\ & & \quad (\lambda (A_0 : \square_i)\, (A_\varnothing : \mathbb{E} \rightarrow A_0).\, A_\varnothing\ e)\ \omega\ A \end{array}$$

[3] The notation El refers to universes à la Tarski in Martin-Löf type theory.

$$\begin{aligned}
[\square_i] &:= \texttt{TypeVal type}_i\ \texttt{TypeErr}_i \\
[x] &:= x \\
[\lambda x : A.\, M] &:= \lambda x : [\![A]\!].\, [M] \\
[M\ N] &:= [M]\ [N] \\
[\Pi x : A.\, B] &:= \texttt{TypeVal}\ (\Pi x : [\![A]\!].\, [\![B]\!])\ (\lambda (e : \mathbb{E})\, (x : [\![A]\!]).\, [B]_\varnothing\ e) \\
[A]_\varnothing &:= \texttt{Err}\ [A] \\
[\![A]\!] &:= \texttt{El}\ [A] \\
[\![\cdot]\!] &:= \cdot \\
[\![\Gamma, x : A]\!] &:= [\![\Gamma]\!], x : [\![A]\!]
\end{aligned}$$

Fig. 2. Exceptional translation

The exceptional translation is defined in Fig. 2. As usual for syntactic translations [9], the term translation is given by $[\cdot]$ and the type translation, written $[\![\cdot]\!]$, is derived from it using the function El. There is an additional macro $[\cdot]_\varnothing$, defined using $\texttt{Err}_i$, which corresponds to the way to inhabit a given type from an exception.

Note that we will often slightly abuse the translation and use the $[\cdot]$ and $[\![\cdot]\!]$ notation as macros acting on the target theory. This is merely for readability purposes, and the corresponding uses are easily expanded to the actual term.

The following lemma makes explicit how $[\![\cdot]\!]$ and $[\cdot]_\varnothing$ behave on universes and on the dependent function space.

Lemma 3 (Unfoldings). *The following definitional equations hold:*

- $[\![\square_i]\!] \equiv \texttt{type}_i$
- $[\![\Pi x : A.\, B]\!] \equiv \Pi x : [\![A]\!].\, [\![B]\!]$
- $[\square_i]_\varnothing\ e \equiv \texttt{TypeErr}_i\ e$
- $[\Pi x : A.\, B]_\varnothing\ e \equiv \lambda x : [\![A]\!].\, [B]_\varnothing\ e$

Proof. By unfolding and straightforward reductions.

The soundness of the translation follows from the following properties, which are fundamental but straightforward to prove.

Theorem 4 (Soundness). *The following properties hold.*

- $[M\{x := N\}] \equiv [M]\{x := [N]\}$ *(substitution lemma).*
- *If* $M \equiv N$ *then* $[M] \equiv [N]$ *(conversion lemma).*
- *If* $\Gamma \vdash M : A$ *then* $[\![\Gamma]\!] \vdash [M] : [\![A]\!]$ *(typing soundness).*
- *If* $\Gamma \vdash A : \square$ *then* $[\![\Gamma]\!] \vdash [A]_\varnothing : \mathbb{E} \to [\![A]\!]$ *(exception soundness).*

Proof. The first property is by routine induction on M, the second is direct by induction on the conversion derivation. The third is by induction on the typing derivation, the most important rule being $\square_i : \square_j$, which holds because $[\square_i] \equiv \texttt{TypeVal}\ \texttt{type}_i\ \texttt{TypeErr}_i$ has type $\texttt{type}_j$ which is convertible to $[\![\square_j]\!]$ by Lemma 3. The last property is a direct application of typing soundness and unfolding of Lemma 3 for universes.

We call $\mathcal{T}_{\mathbb{E}}$ the theory arising from this interpretation, which is formally defined in a way similar to standard categorical constructions over dependent type theory. Terms and contexts of $\mathcal{T}_{\mathbb{E}}$ are simply terms and contexts of $\mathcal{T}$. A context Γ is valid is $\mathcal{T}_{\mathbb{E}}$ whenever its translation $[\![\Gamma]\!]$ is valid in $\mathcal{T}$. Two terms M and N are convertible in $\mathcal{T}_{\mathbb{E}}$ whenever their translations $[M]$ and $[N]$ are convertible in $\mathcal{T}$. Finally, $\Gamma \vdash_{\mathcal{T}_{\mathbb{E}}} M : A$ whenever $[\![\Gamma]\!] \vdash_{\mathcal{T}} [M] : [\![A]\!]$.

That is, it is possible to extend $\mathcal{T}_{\mathbb{E}}$ with a new constant $\texttt{c}$ of a given type A by providing an inhabitant $\texttt{c}_{\mathbb{E}}$ of the translated type $[\![A]\!]$. Then the translation is extended with $[\texttt{c}] := \texttt{c}_{\mathbb{E}}$. The potential computational rules satisfied by this new constant are directly given by the computational rules satisfied by its translation. In some sense, the new constant $\texttt{c}$ is just syntactic sugar for $\texttt{c}_{\mathbb{E}}$. Using $\mathcal{T}_{\mathbb{E}}$, Theorem 4 can be rephrased in the following way.

Theorem 5. *If $\mathcal{T}$ interprets* CC_ω *then so does $\mathcal{T}_{\mathbb{E}}$, that is, the exceptional translation is a syntactic model of* CC_ω*.*

2.2 Exceptional Inductive Types

The fact that the only effect we consider is raising exceptions does not really affect the negative fragment when compared to our previous work [2], but it sure shines when it comes to interpreting inductive datatypes. Indeed, as explained in the introduction, the weaning translation only interprets a subset of CIC, restricting dependent elimination to linear predicates. Furthermore, it also requires a few syntactic properties of the underlying monad ensuring that positivity criteria are preserved through the translation, which can be sometimes hard to obtain.

The exceptional translation diverges from the weaning translation precisely on inductives types. It allows a more compact translation of the latter, while at the same time providing a complete interpretation of CIC, that is, including full dependent elimination.

From now on, we assume that the target theory is a predicative restriction of CIC, i.e. that we can construct in it new inductive datatypes as we do in e.g. Coq [10], but without considering an impredicative universe. That is, all the inductive types we consider in this section live in $\square$. As a matter of fact, we slightly abuse the usual nomenclature and simply call CIC this predicative fragment in the remainder of the paper. We refrain from describing the generic typing rules that extend CC_ω into CIC, as they are fairly standard and would take up too much space. See for instance Werner's thesis for a comprehensive presentation [11].

$$\begin{aligned}
[\mathcal{I}] &:= \lambda(p_1 : [\![P_1]\!]) \;...\; (p_n : [\![P_n]\!])\,(i_1 : [\![I_1]\!]) \;...\; (i_m : [\![I_m]\!]).\\
&\qquad \texttt{TypeVal}\;(\mathcal{I}^\bullet\; p_1 \;...\; p_n\; i_1 \;...\; i_m)\;(\mathcal{I}_\varnothing\; p_1 \;...\; p_n\; i_1 \;...\; i_m)\\
[c_1] &:= c_1^\bullet\\
&\ldots\\
[c_k] &:= c_k^\bullet
\end{aligned}$$

Fig. 3. Inductive type translation

Type and Constructor Translation. As explained before, the intuitive interpretation of a type through the exceptional translation is a pair of a type and a default function from exceptions into that type. In particular, when translating some inductive type $\mathcal{I}$, we must come up with a type $[\![\mathcal{I}]\!]$ together with a default function $\mathbb{E} \to [\![\mathcal{I}]\!]$. As soon as $\mathbb{E}$ is inhabited, that means that we need $[\![\mathcal{I}]\!]$ to be inhabited, preferably in a canonical way. The solution is simple: just as for types where we freely added the exceptional case by means of the `TypeErr` constructor, we freely add exceptions to every inductive type.

In practice, there is an elegant and simple way to do this. It just consists in translating constructors pointwise, while adding a new dedicated constructor standing for the exceptional case. We now turn to the formal construction.

Definition 6. *Let $\mathcal{I}$ be an inductive datatype with*

- *parameters $p_1 : P_1, \ldots, p_n : P_n$;*
- *indices $i_1 : I_1, \ldots, i_m : I_m$;*
- *constructors*
  $c_1 : \Pi(a_{1,1} : A_{1,1}) \ldots (a_{1,l_1} : A_{1,l_1}).\, \mathcal{I}\; p_1 \;\ldots\; p_n\; V_{1,1} \;\ldots\; V_{1,m}$
  $\ldots$
  $c_k : \Pi(a_{k,1} : A_{k,1}) \ldots (a_{k,l_k} : A_{k,l_k}).\, \mathcal{I}\; p_1 \;\ldots\; p_n\; V_{k,1} \;\ldots\; V_{k,m}$

We define the exceptional translation of $\mathcal{I}$ and its constructors in Fig. 3, where $\mathcal{I}^\bullet$ is the inductive type defined by

- *parameters $p_1 : [\![P_1]\!], \ldots, p_n : [\![P_n]\!]$;*
- *indices $i_1 : [\![I_1]\!], \ldots, i_m : [\![I_m]\!]$;*
- *constructors*
  $c_1^\bullet : \Pi(a_{1,1} : [\![A_{1,1}]\!]) \ldots (a_{1,l_1} : [\![A_{1,l_1}]\!]).\, \mathcal{I}^\bullet\; p_1 \;\ldots\; p_n\; [V_{1,1}] \;\ldots\; [V_{1,m}]$
  $\ldots$
  $c_k^\bullet : \Pi(a_{k,1} : [\![A_{k,1}]\!]) \ldots (a_{k,l_k} : [\![A_{k,l_k}]\!]).\, \mathcal{I}^\bullet\; p_1 \;\ldots\; p_n\; [V_{k,1}] \;\ldots\; [V_{k,m}]$
  $\mathcal{I}_\varnothing : \Pi(i_1 : [\![I_1]\!]) \ldots (i_m : [\![I_m]\!]).\, \mathbb{E} \to \mathcal{I}^\bullet\; p_1 \;\ldots\; p_n\; i_1 \;\ldots\; i_m$

where in the recursive calls in the various A, we locally set

$$[\![\mathcal{I}\; M_1 \;\ldots\; M_n\; N_1 \;\ldots\; N_m]\!] := \mathcal{I}^\bullet\; [M_1] \;\ldots\; [M_n]\; [N_1] \;\ldots\; [N_m].$$

Example 7. We give a few representative examples of the inductive translation in Fig. 4 in a Coq-like syntax. They were chosen because they are simple instances of inductive types featuring parameters, indices and recursion in an orthogonal way. For convenience, we write $\Sigma\; A\; (\lambda x : A.\, B)$ as $\Sigma x : A.\, B$.

$$\begin{array}{l||l}
\texttt{Ind bool} : \Box := & \texttt{Ind bool}^\bullet : \Box := \\
\mid \texttt{true} : \texttt{bool} & \mid \texttt{true}^\bullet : \texttt{bool}^\bullet \\
\mid \texttt{false} : \texttt{bool} & \mid \texttt{false}^\bullet : \texttt{bool}^\bullet \\
 & \mid \texttt{bool}_\varnothing : \mathbb{E} \to \texttt{bool}^\bullet \\
\texttt{Ind list}\ (A : \Box) : \Box := & \texttt{Ind list}^\bullet\ (A : [\![\Box]\!]) : \Box := \\
\mid \texttt{nil} : \texttt{list}\ A & \mid \texttt{nil}^\bullet : \texttt{list}^\bullet\ A \\
\mid \texttt{cons} : A \to \texttt{list}\ A \to \texttt{list}\ A & \mid \texttt{cons}^\bullet : [\![A]\!] \to \texttt{list}^\bullet\ A \to \texttt{list}^\bullet\ A \\
 & \mid \texttt{list}_\varnothing : \mathbb{E} \to \texttt{list}^\bullet\ A \\
\texttt{Ind}\ \Sigma\ (A : \Box)\,(B : A \to \Box) : \Box := & \texttt{Ind}\ \Sigma^\bullet\ (A : [\![\Box]\!])\,(B : [\![A]\!] \to \Box) : \Box := \\
\mid \texttt{ex} : \Pi(x : A)\,(y : B\ x).\, \Sigma\ A\ B & \mid \texttt{ex}^\bullet : \Pi(x : [\![A]\!])\,(y : [\![B\ x]\!]).\, \Sigma^\bullet\ A\ B \\
 & \mid \Sigma_\varnothing : \mathbb{E} \to \Sigma^\bullet\ A\ B \\
\texttt{Ind eq}\ (A : \Box)\,(x : A) : A \to \Box := & \texttt{Ind eq}^\bullet\ (A : [\![\Box]\!])\,(x : [\![A]\!]) : [\![A]\!] \to \Box := \\
\mid \texttt{refl} : \texttt{eq}\ A\ x\ x & \mid \texttt{refl}^\bullet : \texttt{eq}^\bullet\ A\ x\ x \\
 & \mid \texttt{eq}_\varnothing : \Pi y : [\![A]\!].\, \mathbb{E} \to \texttt{eq}^\bullet\ A\ x\ y
\end{array}$$

Fig. 4. Examples of translations of inductive types

Remark 8. The fact the we locally override the translation for recursive calls on the $[\![\cdot]\!]$ translation of the type being defined means that we cannot handle cases where the translation of the type of a constructor actually contains an instance of $[\mathcal{I}]$. Because of the syntactic positivity criterion, the only possibility for such a situation to occur in CIC is in the so-called nested inductive definitions. However, nested inductive types are essentially a programming convenience, as most nested types can be rewritten in an isomorphic way that is not nested.

Lemma 9. *If $\mathcal{I}$ is given as in Definition 6, we have for any terms $\vec{M}$, $\vec{N}$*

$$[\![\mathcal{I}\ M_1\ \ldots\ M_n\ N_1\ \ldots\ N_m]\!] \equiv \mathcal{I}^\bullet\ [M_1]\ \ldots\ [M_n]\ [N_1]\ \ldots\ [N_m].$$

This justifies a posteriori the simplified local definition we used in the recursive calls of the translation of the constructors.

Theorem 10. *For any inductive type $\mathcal{I}$ not using nested inductive types, the translation from Definition 6 is well-typed and satisfies the positivity criterion.*

Proof. Preservation of typing is a consequence of Theorem 4. The restriction on nested types, which is slightly stronger than the usual positivity criterion of CIC, is due to the fact that $\mathcal{I}_\varnothing$ is not available in the recursive calls and thus cannot be used to build a term of type `type` via the `TypeVal` constructor.

Preservation of the positivity criterion is straightforward, as the shape of every constructor c_k is preserved, and furthermore by Lemma 3 the structure of every argument type is preserved by $[\![\cdot]\!]$ as well. The only additional constructor $\mathcal{I}_\varnothing$ does not mention the recursive type and is thus automatically positive.

Corollary 11. *Type soundness holds for the translation of inductive types and their constructors.*

Pattern-Matching Translation. We now turn to the translation of the elimination of inductive terms, that is, pattern matching. Once again, its definition originates from the fact that we are working with call-by-name exceptions. It is well-known that in call-by-name, pattern matching implements a delimited form of call-by-value, by forcing its scrutinee before proceeding, at least up to the head constructor. Therefore, as soon as the matched term (re-)raises an exception, the whole pattern-matching reraises the same exception. A little care has to be taken in order to accomodate for the fact that the return type of the pattern-matching depends on the scrutinee, in particular when it is the default constructor of the inductive type.

In what follows, we use the $i_1 \ldots i_n$ notation for clarity, but compact it to $\vec{i}$ for space reasons, when appropriate.

Definition 12. *Assume an inductive $\mathcal{I}$ as given in Definition 6. Let Q be the well-typed pattern-matching defined as*

$$\begin{array}{l}
\texttt{match}\ M\ \texttt{return}\ \lambda(i_1 : I_1)\ldots(i_m : I_m)\,(x : \mathcal{I}\ X_1\ \ldots\ X_n\ i_1\ \ldots\ i_m).\,R\ \texttt{with}\\
\mid\ c_1\ a_{1,1}\ \ldots\ a_{1,l_1}\ \Rightarrow\ N_1\\
\ldots\\
\mid\ c_k\ a_{k,1}\ \ldots\ a_{k,l_k}\ \Rightarrow\ N_k\\
\texttt{end}
\end{array}$$

where

$$\Gamma \vdash \vec{X} : \vec{P} \qquad \Gamma \vdash \vec{Y} : \vec{I}\{\vec{p} := \vec{X}\} \qquad \Gamma \vdash M : \mathcal{I}\ X_1\ \ldots\ X_n\ Y_1\ \ldots\ Y_m$$
$$\Gamma, \vec{i} : \vec{I}\{\vec{p} := \vec{X}\}, x : \mathcal{I}\ \vec{X}\ \vec{i} \vdash R : \square \qquad \Gamma \vdash Q : R\{\vec{i} := \vec{Y}, x := M\}$$
$$\Gamma, \vec{a}_1 : \vec{A}_1 \vdash N_1 : R\{\vec{i} := \vec{V}_1\{\vec{p} := \vec{X}\}, x := c_1\ \vec{X}\ \vec{a}_1\}$$
$$\ldots$$
$$\Gamma, \vec{a}_k : \vec{A}_k \vdash N_k : R\{\vec{i} := \vec{V}_k\{\vec{p} := \vec{X}\}, x := c_k\ \vec{X}\ \vec{a}_k\}$$

then we pose $[Q]$ to be the following pattern-matching.

$$\begin{array}{l}
\texttt{match}\ [M]\ \texttt{return}\ \lambda(i_1 : [\![I_1]\!])\ldots(i_m : [\![I_m]\!])\,(x : \mathcal{I}^\bullet\ [X_1]\ldots[X_n]\ i_1\ldots i_m).\,[\![R]\!]\ \texttt{with}\\
\mid\ c_1^\bullet\ a_{1,1}\ \ldots\ a_{1,l_1}\ \Rightarrow\ [N_1]\\
\ldots\\
\mid\ c_k^\bullet\ a_{k,1}\ \ldots\ a_{k,l_k}\ \Rightarrow\ [N_k]\\
\mid\ \mathcal{I}_\varnothing\ i_1\ \ldots\ i_m\ e\ \Rightarrow\ [R]_\varnothing\{x := \mathcal{I}_\varnothing\ X_1\ \ldots\ X_n\ i_1\ \ldots\ i_m\ e\}\ e\\
\texttt{end}
\end{array}$$

Lemma 13. *With notations and typing assumptions from Definition 12, we have*

$$[\![\Gamma]\!] \vdash [Q] : [\![R]\!]\{\vec{i} := [\vec{Y}], x := [M]\}.$$

Proof. Mostly a consequence of Theorem 4 applied to all of the premises of the pattern-matching rule. The only thing we have to check specifically is that the branch for the default constructor $\mathcal{I}_\varnothing$ is well-typed as

$$[\![\Gamma]\!], \vec{i} : \vec{I}\{\vec{p} := \vec{X}\}, e : \mathbb{E} \vdash [R]_\varnothing\{x := \mathcal{I}_\varnothing\ \vec{X}\ \vec{i}\ e\}\ e : [\![R]\!]\{x := \mathcal{I}_\varnothing\ \vec{X}\ \vec{i}\ e\}$$

which is also due to Theorem 4 applied to R.

Lemma 14. *The translation preserves ι-rules.*

Proof. Immediate, as the translation preserves the structure of the patterns.

The translation is also applicable to fixpoints, but for the sake of readability we do not want to fully spell it out, although it is simply defined by congruence (commutation with the syntax). As such, it trivially preserves typing and reduction rules. Note that the Coq plugin presented in Sect. 6 features a complete translation of inductive types, pattern-matching and fixpoints. So the interested reader may experiment with the plugin to see how fixpoints are translated.

Therefore, by summarizing all of the previous properties, we have the following result.

Theorem 15. *If $\mathcal{T}$ interprets* CIC*, then so does $\mathcal{T}_{\mathbb{E}}$, and thus the exceptional translation is a syntactic model of* CIC*.*

2.3 Flirting with Inconsistency

It is now time to point at the elephant in the room. The exceptional translation has a lot of nice properties, but it has one grave defect.

Theorem 16. *If $\mathbb{E}$ is inhabited, then $\mathcal{T}_{\mathbb{E}}$ is logically inconsistent.*

Proof. The empty type is translated as

$$\texttt{Ind empty}^\bullet \ : \square := \texttt{empty}_\varnothing : \mathbb{E} \rightarrow \texttt{empty}^\bullet$$

which is inhabited as soon as $\mathbb{E}$ is.

Note that when $\mathbb{E}$ is empty, the situation is hardly better, as the translation is essentially the identity. However, when $\mathcal{T}$ satisfies canonicity, the situation is not totally desperate as $\mathcal{T}_{\mathbb{E}}$ enjoys the following weaker canonicity lemma.

Lemma 17 (Exceptional Canonicity). *Let $\mathcal{I}$ be an inductive type with constructors c_1, ..., c_n and assume that $\mathcal{T}$ satisfies canonicity. The translation of any closed term $\vdash_{\mathcal{T}_{\mathbb{E}}} M : \mathcal{I}$ evaluates either to a constructor of the form $c_i^\bullet\ N_1 \ldots N_{l_i}$ or to the default constructor $\mathcal{I}_\varnothing\ e$ for some $e : \mathbb{E}$.*

Proof. Direct application of Theorem 4 and canonicity of $\mathcal{T}$.

A direct consequence of Lemma 17 is that any proof of the empty type is an exception. As we will see in Sect. 4.1, for some types it is also possible to dynamically check whether a term of this type is a correct proof, in the sense that it does not raise an uncaught exception. This means that while $\mathcal{T}_{\mathbb{E}}$ is logically unsound, it is computationally relevant and can still be used as a *dependently-typed programming language with exceptions*, a shift into a realm where we would have called the weaker canonicity Lemma 17 a *progress lemma*.

This is not the end of the story, though. Recall that $\mathcal{T}_{\mathbb{E}}$ only exists through its embedding $[\cdot]$ into $\mathcal{T}$. In particular, if $\mathcal{T}$ is consistent, this means that one can reason about terms of $\mathcal{T}_{\mathbb{E}}$ directly in $\mathcal{T}$. For instance, it is possible to prove in $\mathcal{T}$ that assuming some properties about its input, a function in $\mathcal{T}_{\mathbb{E}}$ never raises an exception. Hence not only do we have an effectul programming language, but we also have a *sound logical framework* allowing to transparently prove safety properties about impure programs.

It is actually even better than that. We will show in Sect. 3 that safety properties can be derived automatically for pure programs, allowing to recover a consistent type theory as long as $\mathcal{T}$ is consistent itself.

2.4 Living in an Exceptional World

We describe here what $\mathcal{T}_{\mathbb{E}}$ feels like in direct style. The exceptional theory feature a new type $\mathbf{E}$ which reifies the underlying type $\mathbb{E}$ of exceptions in $\mathcal{T}_{\mathbb{E}}$. It uses the fact that for $\mathbb{E}$, the default function (here of type $\mathbb{E} \to \mathbb{E}$) can simply be defined as the identity function. Its translation is given by

$$[\mathbf{E}] : [\![\Box]\!] := \texttt{TypeVal}\ \mathbb{E}\ (\lambda e : \mathbb{E}.\, e).$$

Then, it is possible to define in $\mathcal{T}_{\mathbb{E}}$ a function $\texttt{raise} : \Pi A : \Box.\, \mathbf{E} \to A$ that raises the provided exception at any type as

$$[\texttt{raise}] := \lambda (A : \texttt{type})\, (e : \mathbb{E}).\, \texttt{Err}\ A\ e.$$

As we have already mentioned, the reader should be aware that the exceptions arising from this translation are call-by-name. This means that they do not behave like their usual call-by-value counterpart. In particular, we have in $\mathcal{T}_{\mathbb{E}}$

$$\texttt{raise}\ (\Pi x : A.\, B)\ e \equiv \lambda x : A.\, \texttt{raise}\ B\ e$$

which means that exceptions cannot be caught on Π-types. We can catch them on universes and inductive types though, because in those cases they are freely added through an extra constructor which one can pattern-match on. For instance, there exists in $\mathcal{T}_{\mathbb{E}}$ a term

$$\begin{aligned}\texttt{catch}_{\texttt{bool}} : \Pi P : \texttt{bool} \to \Box.\, P\ \texttt{true} \to P\ \texttt{false} \to&\\ (\Pi e : \mathbf{E}.\, P\ (\texttt{raise}\ \texttt{bool}\ e)) \to \Pi b : \texttt{bool}.\, P\ b&\end{aligned}$$

defined by

$$\begin{aligned}[\texttt{catch}_{\texttt{bool}}] := \lambda P\, p_t\, p_f\, p_e\, b.\ \texttt{match}\ b\ \texttt{return}\ \lambda b.\, \texttt{El}\ (P\ b)\ \texttt{with}&\\ \mid \texttt{true}^\bullet \Rightarrow p_t&\\ \mid \texttt{false}^\bullet \Rightarrow p_f&\\ \mid \texttt{bool}_\varnothing\ e \Rightarrow p_e\ e&\\ \texttt{end}&\end{aligned}$$

satisfying the expected reduction rules on all three cases.

In Sect. 6, we illustrate the use of the exceptional theory using the Coq plugin to define a simple cast framework as in [12].

$$
\begin{array}{ll}
[\square_i]_\varepsilon & := \lambda A : [\![\square_i]\!].\, [\![A]\!] \to \square_i \\
[x]_\varepsilon & := x_\varepsilon \\
[\lambda x : A.\, M]_\varepsilon & := \lambda(x : [\![A]\!])\,(x_\varepsilon : [\![A]\!]_\varepsilon\ x).\, [M]_\varepsilon \\
[M\ N]_\varepsilon & := [M]_\varepsilon\ [N]\ [N]_\varepsilon \\
[\Pi x : A.\, B]_\varepsilon & := \lambda(f : \Pi x : [\![A]\!].\, [\![B]\!]).\, \Pi(x : [\![A]\!])\,(x_\varepsilon : [\![A]\!]_\varepsilon\ x).\, [\![B]\!]_\varepsilon\ (f\ x) \\
[\![A]\!]_\varepsilon & := [A]_\varepsilon \\
[\![\cdot]\!]_\varepsilon & := \cdot \\
[\![\Gamma, x : A]\!]_\varepsilon & := [\![\Gamma]\!]_\varepsilon, x : [\![A]\!], x_\varepsilon : [\![A]\!]_\varepsilon\ x
\end{array}
$$

Fig. 5. Parametricity over exceptional translation

3 Kreisel Meets Martin-Löf

It is well-known that Reynolds' parametricity [13] and Kreisel's modified realizability [4] are two instances of the broader logical relation techniques. Usually, parametricity is used to derive theorems for free, while realizability constrains programs. In a surprising turn of events, we use Bernardy's variant of parametricity on CIC [5] as a realizability trick to evict undesirable behaviours of $\mathcal{T}_{\mathbb{E}}$. This leads to the *parametric exceptional translation*, which can be seen as the embodiment of Kreisel's realizability in type theory. In this section, we first present this translation on the negative fragment, then extend it to CIC and finally discuss its meta-theoretical properties.

3.1 Exceptional Parametricity in a Negative World

The exceptional parametricity translation for terms of CC_ω is defined in Fig. 5. Intuitively, any type A in $\mathcal{T}_{\mathbb{E}}$ is turned into a validity predicate $A_\varepsilon : A \to \square$ which encodes the fact that an inhabitant of A is not allowed to generate unhandled exceptions. For instance, a function is valid if its application to a valid term produces a valid answer. It does not say anything about the application to invalid terms though, which amounts to a *garbage in, garbage out* policy. The translation then states that every pure term is automatically valid.

This translation is exactly standard parametricity for type theory [5] but parametrized by the exceptional translation. This means that any occurrence of a term of the original theory used in the parametricity translation is replaced by its exceptional translation, using $[\cdot]$ or $[\![\cdot]\!]$ depending on whether it is used as a term or as a type. For instance, the translation of an application $[M\ N]_\varepsilon$ is given by $[M]_\varepsilon\ [N]\ [N]_\varepsilon$ instead of just $[M]_\varepsilon\ N\ [N]_\varepsilon$.

Lemma 18 (Substitution lemma). *The translation satisfies the following conversion:* $[M\{x := N\}]_\varepsilon \equiv [M]_\varepsilon\{x := [N], x_\varepsilon := [N]_\varepsilon\}$.

Theorem 19 (Soundness). *The two following properties hold.*

– *If* $M \equiv N$ *then* $[M]_\varepsilon \equiv [N]_\varepsilon$.
– *If* $\Gamma \vdash M : A$ *then* $[\![\Gamma]\!]_\varepsilon \vdash [M]_\varepsilon : [\![A]\!]_\varepsilon\ [M]$.

Proof. By induction on the derivation.

We can use this result to construct another syntactic model of CC_ω. Contrarily to usual syntactic models where sequents are straightforwarldy translated to sequents, this model is slightly more subtle as sequents are translated to pairs of sequents instead. This is similar to the usual parametricity translation.

Definition 20. *The theory* $\mathcal{T}^p_{\mathbb{E}}$ *is defined by the following data.*

– *Terms of* $\mathcal{T}^p_{\mathbb{E}}$ *are pairs of terms of* $\mathcal{T}$.
– *Contexts of* $\mathcal{T}^p_{\mathbb{E}}$ *are pairs of contexts of* $\mathcal{T}$.
– $\vdash_{\mathcal{T}^p_{\mathbb{E}}} \Gamma$ *whenever* $\vdash_{\mathcal{T}} [\![\Gamma]\!]$ *and* $\vdash_{\mathcal{T}} [\![\Gamma]\!]_\varepsilon$.
– $M \equiv_{\mathcal{T}^p_{\mathbb{E}}} N$ *whenever* $[M] \equiv_{\mathcal{T}} [N]$ *and* $[M]_\varepsilon \equiv_{\mathcal{T}} [N]_\varepsilon$.
– $\Gamma \vdash_{\mathcal{T}^p_{\mathbb{E}}} M : A$ *whenever* $[\![\Gamma]\!] \vdash_{\mathcal{T}} [M] : [\![A]\!]$ *and* $[\![\Gamma]\!]_\varepsilon \vdash_{\mathcal{T}} [M]_\varepsilon : [\![A]\!]_\varepsilon\ [M]$.

Once again, Theorem 19 can be rephrased in terms of preservation of theories and syntactic models.

Theorem 21. *If* $\mathcal{T}$ *interprets* CC_ω *then so does* $\mathcal{T}^p_{\mathbb{E}}$. *That is, the parametric exceptional translation is a syntactic model of* CC_ω.

This construction preserves definitional η-expansion, as functions are mapped to (slightly more complicated) functions.

Lemma 22. *If* $\mathcal{T}$ *satisfies definitional* η*-expansion, then so does* $\mathcal{T}^p_{\mathbb{E}}$.

Proof. The first component of the translation preserves definitional η-expansion because functions are mapped to functions. It remains to show that

$$[\lambda x : A.\, M\ x]_\varepsilon := \lambda(x : [\![A]\!])\,(x_\varepsilon : [\![A]\!]_\varepsilon\ x).\, [M]_\varepsilon\ x\ x_\varepsilon \equiv [M]_\varepsilon$$

which holds by applying η-expansion twice.

It is interesting to remark that Bernardy-style unary parametricity also leads to a syntactic model $\mathcal{T}^p$ that interprets CC_ω (as well as CIC), using the same kind of glueing construction. Nonetheless, this model is somewhat degenerate from the logical point of view. Namely it is a conservative extension of the target theory. Indeed, if $\Gamma \vdash_{\mathcal{T}^p} M : A$ for some Γ, M and A from $\mathcal{T}$, then there we also have $\Gamma \vdash_{\mathcal{T}} M : A$, because the first component of the model is the identity, and the original sequent can be retrieved by the first projection.

This is definitely *not* the case with the $\mathcal{T}^p_{\mathbb{E}}$ theory, because the first projection is not the identity. In particular, because of Theorem 16, every sequent in the first projection is inhabited, although it is not the case in $\mathcal{T}$ itself if it is consistent. This means that parametricity can actually bring additional expressivity when it applies to a theory which is not pure, as it is the case here.

$$
\begin{array}{l}
\texttt{Ind bool}_\varepsilon\ : \texttt{bool}^\bullet \to \square := \\
\mid \texttt{true}_\varepsilon : \texttt{bool}_\varepsilon\ \texttt{true}^\bullet \\
\mid \texttt{false}_\varepsilon : \texttt{bool}_\varepsilon\ \texttt{false}^\bullet \\[1ex]
\texttt{Ind list}_\varepsilon\ (A : \texttt{type})\,(A_\varepsilon : [\![A]\!] \to \square)\ : \texttt{list}^\bullet\ A \to \square := \\
\mid \texttt{nil}_\varepsilon : \texttt{list}_\varepsilon\ A\ A_\varepsilon\ (\texttt{nil}^\bullet\ A) \\
\mid \texttt{cons}_\varepsilon : \Pi(x : [\![A]\!])\,(x_\varepsilon : A_\varepsilon\ x)\,(l : \texttt{list}^\bullet\ A)\,(l_\varepsilon : \texttt{list}_\varepsilon\ A\ A_\varepsilon\ l). \\
\qquad\qquad \texttt{list}_\varepsilon\ A\ A_\varepsilon\ (\texttt{cons}^\bullet\ A\ x\ l) \\[1ex]
\texttt{Ind eq}_\varepsilon\ (A : \texttt{type})\,(A_\varepsilon : [\![A]\!] \to \square)\,(x : [\![A]\!])\,(x_\varepsilon : A_\varepsilon\ x)\ : \\
\quad \Pi(y : [\![A]\!])\,(y_\varepsilon : A_\varepsilon\ y).\,\texttt{eq}^\bullet\ A\ x\ y \to \square := \\
\mid \texttt{refl}_\varepsilon : \texttt{refl}_\varepsilon\ A\ A_\varepsilon\ x\ x_\varepsilon\ x\ x_\varepsilon\ (\texttt{refl}^\bullet\ A\ x)
\end{array}
$$

Fig. 6. Examples of parametric translation of inductive types

3.2 Exceptional Parametric Translation of CIC

We now describe the parametricity translation of the positive fragment. The intuition is that as it stands for an exception, the default constructor is always invalid, while all other constructors are valid, assuming their arguments are.

Type and Constructor Translation

Definition 23. *Let $\mathcal{I}$ be an inductive type as given in Definition 6. We define the exceptional parametricity translation $\mathcal{I}_\varepsilon$ of $\mathcal{I}$ as the inductive type defined by:*

- *parameters* $[\![p_1 : P_1, \ldots, p_n : P_n]\!]_\varepsilon$*;*
- *indices* $[\![i_1 : I_1, \ldots, i_m : I_m]\!]_\varepsilon, x : \mathcal{I}\ p_1\ \ldots\ p_n\ i_1\ \ldots\ i_m$*;*
- *constructors*

$$
\begin{array}{l}
c_{1\varepsilon} : \Pi[\![\vec{a}_1 : \vec{A}_1]\!]_\varepsilon. \\
\qquad \mathcal{I}_\varepsilon\ p_1\ p_{1\varepsilon}\ \ldots\ p_n\ p_{n\varepsilon}\ [V_{1,1}]\ [V_{1,1}]_\varepsilon\ \ldots\ [V_{1,m}]\ [V_{1,m}]_\varepsilon\ (c_1^\bullet\ \vec{p}\ \vec{a}_1) \\
\ldots \\
c_{k\varepsilon} : \Pi[\![\vec{a}_k : \vec{A}_k]\!]_\varepsilon. \\
\qquad \mathcal{I}_\varepsilon\ p_1\ p_{1\varepsilon}\ \ldots\ p_n\ p_{n\varepsilon}\ [V_{k,1}]\ [V_{k,1}]_\varepsilon\ \ldots\ [V_{k,m}]\ [V_{k,m}]_\varepsilon\ (c_k^\bullet\ \vec{p}\ \vec{a}_k).
\end{array}
$$

and we extend the translation as

$$[\mathcal{I}]_\varepsilon := \mathcal{I}_\varepsilon \quad [c_1]_\varepsilon := c_{1\varepsilon} \quad \ldots \quad [c_k]_\varepsilon := c_{k\varepsilon}.$$

Example 24. We give the exceptional parametric inductive translation of our running examples in Fig. 6.

Note that contrarily to the negative case, the exceptional parametricity translation on inductive types is *not* the same thing as the composition of Bernardy's parametricity together with the exceptional translation. Indeed, the latter would also have produced a constructor for the default case from the exceptional inductive translation, whereas our goal is precisely to rule this case out via the additional realizability-like interpretation.

It is also very different from our previous parametric weaning translation [2], which relies on internal parametricity to recover dependent elimination, enforcing by construction that no effectful term exists. Here, effectful terms may be used in the first component, but they are required after the fact to have no inconsistent behaviour. Intuitively, parametric weaning produces one pure sequent, while exceptional parametricity produces two, with the first one being potentially impure and the second one assuring the first one is harmless.

Pattern-Matching Translation

Definition 25. *Let Q be the pattern-matching defined in Definition 12. We pose $[Q]_\varepsilon$ to be the pattern-matching*

$$
\begin{array}{l}
\texttt{match}\ [M]_\varepsilon\ \texttt{return}\ \lambda[\![\vec{i} : \vec{I}]\!]_\varepsilon\,(x : \mathcal{I}^\bullet\ [X_1]\ \ldots\ [X_n]\ i_1\ \ldots\ i_m).\\
\qquad\qquad\qquad\qquad\qquad (x_\varepsilon : \mathcal{I}_\varepsilon\ [X_1]\ [X_1]_\varepsilon\ \ldots\ [X_n]\ [X_n]_\varepsilon\ i_1\ i_{1\varepsilon}\ \ldots\ i_m\ i_{m\varepsilon}\ x)\\
\qquad\qquad\qquad\qquad\qquad\quad [\![R]\!]_\varepsilon\ [Q_x]\\
\texttt{with}\\
\mid\ c_{1\varepsilon}\ a_{1,1}\ a_{1,1\varepsilon}\ \ldots\ a_{1,l_1}\ a_{1,l_1\varepsilon}\ \Rightarrow\ [N_1]_\varepsilon\\
\ldots\\
\mid\ c_{k\varepsilon}\ a_{k,1}\ a_{k,1\varepsilon}\ \ldots\ a_{k,l_k}\ a_{1,l_k\varepsilon}\ \Rightarrow\ [N_k]_\varepsilon\\
\texttt{end}
\end{array}
$$

where Q_x is the following pattern-matching

$$
\begin{array}{l}
\texttt{match}\ x\ \texttt{return}\ \lambda(i_1 : I_1)\ldots(i_m : I_m)\,(x : \mathcal{I}\ X_1\ \ldots\ X_n\ i_1\ \ldots\ i_m).\,R\ \texttt{with}\\
\mid\ c_1\ a_{1,1}\ \ldots\ a_{1,l_1}\ \Rightarrow\ N_1\\
\ldots\\
\mid\ c_k\ a_{k,1}\ \ldots\ a_{k,l_k}\ \Rightarrow\ N_k\\
\texttt{end}
\end{array}
$$

that is Q where the scrutinee has been turned into the index variable of the parametricity predicate.

Lemma 26. *With notations and typing assumptions from Definition 12, we have*

$$[\![\Gamma]\!]_\varepsilon \vdash [Q]_\varepsilon : [\![R\{\vec{i} := \vec{Y}, x := M\}]\!]_\varepsilon\ [Q].$$

The exceptional parametricity translation can be extended to handle fixpoints as well, with a few limitations. Translating generic fixpoints uniformly is indeed an open problem in standard parametricity, and our variant faces the same issue. In practice, standard recursors can be automatically translated, and fancy fixpoints may require hand-writing the parametricity proof. We do not describe the recursor translation here though, as it is essentially the same as standard parametricity. Again, the interested reader may test the Coq plugin exposed in Sect. 6 to see how recursors are translated.

Packing everything together allows to state the following result.

Theorem 27. *If $\mathcal{T}$ interprets* CIC, *then so does $\mathcal{T}^p_{\mathbb{E}}$, and thus the exceptional parametricity translation is a syntactic model of* CIC.

3.3 Meta-Theoretical Properties of $\mathcal{T}_{\mathbb{E}}^{p}$

Being built as a syntactic model, $\mathcal{T}_{\mathbb{E}}^{p}$ inherits a lot of meta-theoretical properties of $\mathcal{T}$. We list a few of interest below.

Theorem 28. *If $\mathcal{T}$ is consistent, then so is $\mathcal{T}_{\mathbb{E}}^{p}$.*

Proof. Assume $\vdash_{\mathcal{T}_{\mathbb{E}}^{p}} M_0 : \texttt{empty}$ for some M_0. Then by definition, there exists two terms M and M_ε such that $\vdash_{\mathcal{T}} M : \texttt{empty}^\bullet$ and $\vdash_{\mathcal{T}} M_\varepsilon : \texttt{empty}_\varepsilon\ M$. But $\texttt{empty}_\varepsilon$ has no constructor, and $\mathcal{T}$ is inconsistent.

More generally, the same argument holds for any inductive type.

Theorem 29. *If $\mathcal{T}$ enjoys canonicity, then so does $\mathcal{T}_{\mathbb{E}}^{p}$.*

Proof. The exceptional parametricity translation for inductive types has the same structure as the original type, so any normal form in $\mathcal{T}_{\mathbb{E}}^{p}$ can be mapped back to a normal form in $\mathcal{T}$.

4 Effectively Extending CIC

The parametric exceptional translation allows to extend the logical expressivity of CIC in the following ways, which we develop in the remainder of this section.

We show in Sect. 4.1 that Markov's rule is admissible in CIC. We already sketched this result in our previous paper [2], but we come back to it in more details. More generally, we show a form of conservativity of double-negation elimination over the type-theoretic version of Π_2^0 formulae.

In Sect. 4.2, we exhibit a syntactic model of CIC which satisfies definitional η-expansion for functions but which negates function extensionality. As far as we know, this was not known.

Finally, in Sect. 4.3, we show that there exists a model of CIC which validates the independence of premises. This is a new result, that shows that CIC can feature traces of classical reasoning while staying computational. We use this result in Sect. 4.4 to give an alternative proof of the recent result of Coquand and Mannaa [7] that Markov's principle is not provable in CIC.

4.1 Markov's Rule

We show in this section that CIC is closed under a generalized Markov's rule. The technique used here is no more than a dependently-typed variant of Friedman's trick [14]. Indeed, Friedman's A-translation amounts to add exceptions to intuitionistic logic, which is precisely what $\mathcal{T}_{\mathbb{E}}$ does for CIC.

Definition 30. *An inductive type in* CIC *is said to be first-order if all the types of the arguments of its constructors, in its parameters and in its indices are recursively first-order.*

Example 31. The `empty`, `unit` and $\mathbb{N}$ types are first-order. If P and Q are first-order then so is $\Sigma p : P. Q$, $P + Q$ and $\texttt{eq}\ P\ p_0\ p_1$. Consequently, the CIC equivalent of Σ_1^0 formulae are in particular first-order.

First-order types enjoy uncommon properties, like the fact that they can be injected into effectful terms and purified away. This is then used to prove the generalized Markov's Rule.

Lemma 32. *For every first-order type* $\vec{p} : \vec{P} \vdash Q : \square$ *where all* $\vec{P}$ *are first-order, there are retractions* $\iota_{\vec{P}}$, ι_Q *and* $\theta_{\vec{P}}$, θ_Q *s.t.:*

$$\vec{p} : \vec{P} \vdash \iota_Q : Q \to [\![Q]\!]\{\vec{p} := \iota_{\vec{P}}\ \vec{p}\}$$
$$\vec{p} : \vec{P} \vdash \theta_Q : [\![Q]\!]\{\vec{p} := \iota_{\vec{P}}\ \vec{p}\} \to Q + \mathbb{E}.$$

Proof. The ι terms exist because effectful inductive types are a semantical superset of their pure equivalent, and the θ terms are implemented by recursively forcing the corresponding impure inductive term. One relies on decidability of equality of first-order type to fix the indices.

Theorem 33 (Generalized Markov's Rule). *For any first-order type* P *and first-order predicate* Q *over* P*, if* $\vdash_{\text{CIC}} \Pi p : P. \neg\neg\ (Q\ p)$ *then* $\vdash_{\text{CIC}} \Pi p : P. Q\ p$.

Proof. Let $\vdash M : \Pi p : P. \neg\neg\ (Q\ p)$. By taking $\mathbb{E} := Q\ p$ and apply the soundness theorem, one gets a proof

$$p : P \vdash [M] : \Pi\hat{p} : [\![P]\!]. ([\![Q\ \hat{p}]\!] \to \texttt{empty}^\bullet) \to \texttt{empty}^\bullet.$$

But $\texttt{empty}^\bullet \cong \mathbb{E} \equiv Q\ p$, so we can derive from $[M]$ a term $M^\sharp$ s.t.

$$p : P \vdash M^\sharp : \Pi\hat{p} : [\![P]\!]. ([\![Q\ \hat{p}]\!] \to Q\ p + Q\ p) \to Q\ p.$$

The proofterm we were looking for is thus no more than $\lambda p : P. M^\sharp\ (\iota_P\ p)\ \theta_Q$.

4.2 Function Intensionality with η-expansion

In a previous paper [9], we already showed that there existed a syntactic model of CIC that allowed to internally disprove function extensionality. Yet, this model was clearly not preserving definitional η-expansion on functions, as it was adding additional structure to abstraction and application (namely a boolean). Thanks to our new model, we can now demonstrate that counterintuitively, it is possible to have a consistent type theory that enjoys definitional η-expansion while negating internally function extensionality. In this section we suppose that $\mathbb{E} := \texttt{unit}$, although any inhabited type of exceptions would work.

By Lemma 22, we know that the parametric exceptional translation preserves definitional η-expansion. It is thus sufficient to find two functions that are extensionally equal but intensionally distinct in the model. Let us consider to this end the $\texttt{unit} \to \texttt{unit}$ functions

$$\texttt{id}_\perp := \lambda u : \texttt{unit}. u \qquad\qquad \texttt{id}_\top := \lambda u : \texttt{unit}. \texttt{tt}.$$

Theorem 34. *The following sequents are derivable:*

$$\vdash_{\mathcal{T}_{\mathbb{E}}^p} \Pi u : \texttt{unit}.\, \texttt{id}_\bot\ u = \texttt{id}_\top\ u \qquad \vdash_{\mathcal{T}_{\mathbb{E}}^p} \texttt{id}_\bot = \texttt{id}_\top \to \texttt{empty}.$$

Proof. The main difference between the two functions is that $\texttt{id}_\bot$ preserves exceptions while $\texttt{id}_\top$ does not, which we exploit.

The first sequent is provable in CIC by dependent elimination and thus is derivable in $\mathcal{T}_{\mathbb{E}}^p$ by applying the soundness theorem.

To prove the first component of the second sequent, we exhibit a property that discriminates $[\texttt{id}_\bot]$ and $[\texttt{id}_\top]$, which is, as explained, their evaluation on the term $\texttt{unit}_\varnothing\ \texttt{tt}$. Showing then that this proof is parametric is equivalent to showing $\Pi(p : [\![\texttt{id}_\bot = \texttt{id}_\top]\!])\,(p_\varepsilon : [\![\texttt{id}_\bot = \texttt{id}_\top]\!]_\varepsilon\ p).\,\texttt{empty}$. But p_ε actually implies $[\texttt{id}_\bot] = [\texttt{id}_\top]$, which we just showed was absurd.

4.3 Independence of Premise

Independence of premise (IP) is a semi-classical principle from first-order logic whose CIC equivalent can be stated as follows.

$$\Pi(A : \square)\,(B : \mathbb{N} \to \square).\,(\neg A \to \Sigma n : \mathbb{N}.\,B\ n) \to \Sigma n : \mathbb{N}.\,\neg A \to B\ n \quad \text{(IP)}$$

Although not derivable in intuitionistic logic, it is an admissible rule of **HA**. The standard proof of this property is to go through Kreisel's modified realizability interpretation of **HA** [4]. In a nutshell, the interpretation goes as follows: by induction over a formula A, define a simple type $\tau(A)$ of realizers of A together with a realizability predicate $\cdot \Vdash A$ over $\tau(A)$. Then show that whenever $\vdash_{\mathbf{HA}} A$, there exists some simply-typed term $t : \tau(A)$ s.t. $t \Vdash A$. As the interpretation also implies that there is no t s.t. $t \Vdash \bot$, this gives a sound model of **HA**, which contains more than the latter. Most notably, there is for instance a term `ip` s.t.

$$\texttt{ip} \Vdash (\neg A \to \exists n.\,B) \to \exists n.\,\neg A \to B$$

for any A, B. Intriguingly, the computational content of `ip` did not seem to receive a fair treatment in the literature. To the best of our knowledge, it has never been explicitly stated that IP was realizable because of the following "bug" of Kreisel's modified realizability.

Lemma 35 (Kreisel's bug). *For every formula A, $\tau(A)$ is inhabited. In particular, $\tau(\bot) := \texttt{unit}$.*

We show that this is actually not a bug, but a hidden feature of Kreisel's modified realizability, which secretly allows to encode exceptions in the realizers. To this end, we implement IP in $\mathcal{T}_{\mathbb{E}}^p$ by relying internally on *paraproofs*, i.e. terms raising exceptions, while ensuring these exceptions never escape outside of the locally unsafe boundary. The resulting $\mathcal{T}_{\mathbb{E}}^p$ term has essentially the same computational content as its Kreisel's realizability counterpart. In this section we suppose that $\mathbb{E} := \texttt{unit}$, although assuming $\mathbb{E}$ to be inhabited is sufficient.

To ease the understanding of the definition, we rely on effectful combinators that can be defined in $\mathcal{T}_{\mathbb{E}}$.

Definition 36. *We define in $\mathcal{T}_{\mathbb{E}}$ the following terms.*

$$\begin{array}{l} \texttt{fail} \;\; : \;\; \Pi A : \square .\, A \\ [\texttt{fail}] := \lambda A : [\![\square]\!].\, [A]_{\varnothing}\ \texttt{tt} \end{array}$$

$$\begin{array}{ll} \texttt{is}_{\Sigma} \;\; : \;\; \Pi A\, B.\, (\Sigma x : A.\, B) \to \texttt{bool} & \texttt{is}_{\mathbb{N}} \;\; : \;\; \mathbb{N} \to \texttt{bool} \\ [\texttt{is}_{\Sigma}] := \lambda A\, B\, p.\, \texttt{match}\ p\ \texttt{with} & [\texttt{is}_{\mathbb{N}}] := \texttt{fix}\ \texttt{is}_{\mathbb{N}}\ n\ := \ \texttt{match}\ n\ \texttt{with} \\ \qquad\qquad\quad \mid \texttt{ex}^{\bullet}\ \_\ \_ \Rightarrow \texttt{true}^{\bullet} & \qquad\qquad \mid \texttt{O}^{\bullet} \Rightarrow \texttt{true}^{\bullet} \\ \qquad\qquad\quad \mid \Sigma_{\varnothing}\ \_ \Rightarrow \texttt{false}^{\bullet} & \qquad\qquad \mid \texttt{S}^{\bullet}\ n \Rightarrow \texttt{is}_{\mathbb{N}}\ n \\ \qquad\qquad\quad \texttt{end} & \qquad\qquad \mid \mathbb{N}_{\varnothing}\ \_ \Rightarrow \texttt{false}^{\bullet} \\ & \qquad\qquad \texttt{end} \end{array}$$

It is worth insisting that these combinators are not necessarily parametric. While it can be shown that $\texttt{is}_{\Sigma}$ and $\texttt{is}_{\mathbb{N}}$ actually are, `fail` is luckily not. The $\texttt{is}_{\Sigma}$ and $\texttt{is}_{\mathbb{N}}$ functions are used in order to check that a value is actually pure and does not contain exceptions.

Definition 37. *We define* `ip` *in $\mathcal{T}_{\mathbb{E}}$ in direct style below, using the available combinators from Definition 36 and a bit of syntactic sugar.*

$$\begin{array}{l} \texttt{ip} : \;\; \mathrm{IP} \\ \texttt{ip} := \lambda(A : \square)\, (B : \mathbb{N} \to \square)\, (f : \neg A \to \Sigma n : \mathbb{N}.\, B\ n). \\ \qquad\quad \texttt{let}\ p\ :=\ f\ (\texttt{fail}\ (\neg A))\ \texttt{in} \\ \qquad\quad \texttt{if}\ \texttt{is}_{\Sigma}\ \mathbb{N}\ B\ p\ \texttt{then}\ \texttt{match}\ p\ \texttt{with} \\ \qquad\quad \mid\ \texttt{ex}\ n\ b\ \Rightarrow\ \texttt{if}\ \texttt{is}_{\mathbb{N}}\ n\ \texttt{then}\ \texttt{ex}\ \_\ \_\ n\ (\lambda \_ : \neg A.\, b) \\ \qquad\qquad\qquad\qquad\qquad\quad \texttt{else}\ \texttt{ex}\ \_\ \_\ \texttt{O}\ (\texttt{fail}\ (\neg A \to B\ \texttt{O})) \\ \qquad\quad \texttt{end}\ \texttt{else}\ \texttt{ex}\ \_\ \_\ \texttt{O}\ (\texttt{fail}\ (\neg A \to B\ \texttt{O})) \end{array}$$

The intuition behind this term is the following. Given $f : \neg A \to \Sigma n : \mathbb{N}.\, B\ n$, we apply it to a dummy function which fails whenever it is used. Owing to the semantics of negation, we know *in the parametricity layer* that the only way for this application to return an exception is that f actually contained a proof of A and applied `fail` to it. Therefore, given a true proof of $\neg A$, we are in an inconsistent setting and thus we are able to do whatever pleases us. The issue is that we do not have access to such a proof yet, and we do have to provide a valid integer now. Therefore, we check whether f actually provided us with a valid pair containing a valid integer. If so, this is our answer, otherwise we stuff a dummy integer value and we postpone the contradiction.

This is essentially the same realizer as the one from Kreisel's modified realizability, except that we have a fancy type system for realizers. In particular, because we have dependent types, integers also exist in the logical layer, so that they need to be checked for exceptions as well. The only thing that remains to be proved is that `ip` also lives in $\mathcal{T}_{\mathbb{E}}^{p}$.

Theorem 38. *There is a proof of* $\vdash_{\mathcal{T}} [\![\mathrm{IP}]\!]_{\varepsilon}\ [\texttt{ip}]$.

Proof. The proof is straightforward but tedious, so we do not give the full details. The file `IPc.v` of the companion Coq plugin contains an explicit proof. The essential properties that make it go through are the following.

- $\vdash_{\mathcal{T}} \Pi(n : \mathbb{N}^{\bullet})\,(p_1\,p_2 : \mathbb{N}_{\varepsilon}\ n).\, p_1 = p_2$
- $\vdash_{\mathcal{T}} \Pi n : \mathbb{N}^{\bullet}.\,[\mathtt{is}_{\mathbb{N}}]\ n = \mathtt{true}^{\bullet} \leftrightarrow \mathbb{N}_{\varepsilon}\ n$
- $\vdash_{\mathcal{T}} \Pi(p\,q : [\![\neg A]\!]).\, [\![\neg A]\!]_{\varepsilon}\ p \to [\![\neg A]\!]_{\varepsilon}\ q$

Corollary 39. *We have* $\vdash_{\mathcal{T}^p_{\mathbb{E}}}$ IP.

4.4 Non-provability of Markov's Principle

From this result, one can get a very easy syntactic proof of the independence result of Markov's principle from CIC. Markov's principle is usually stated as

$$\Pi P : \mathbb{N} \to \mathtt{bool}.\, \neg\neg\ (\Sigma n : \mathbb{N}.\, P\ n = \mathtt{true}) \to \Sigma n : \mathbb{N}.\, P\ n = \mathtt{true} \quad \text{(MP)}$$

An independence result was recently proved by Coquand and Mannaa by a semantic argument [7]. We leverage instead a property from realizability [15] that has been applied to type theory the other way around by Herbelin [16].

Lemma 40. *If $\mathcal{S}$ is a computable theory containing* CIC *and enjoying canonicity, then one cannot have both* $\vdash_{\mathcal{S}}$ IP *and* $\vdash_{\mathcal{S}}$ MP.

Proof. By applying IP to MP, one easily obtains that

$$\vdash_{\mathcal{S}} \Pi P : \mathbb{N} \to \mathtt{bool}.\, \Sigma n : \mathbb{N}.\, \Pi m : \mathbb{N}.\, P\ m = \mathtt{true} \to P\ n = \mathtt{true}.$$

Thus, for every closed $P : \mathbb{N} \to \mathtt{bool}$, by canonicity there exists a closed $n_P : \mathbb{N}$ s.t. $\vdash_{\mathcal{S}} \Pi m : \mathbb{N}.\, P\ m = \mathtt{true} \to P\ n_P = \mathtt{true}$. But then one can decide whether P holds for some n by just computing $P\ n_P$, so that we effectively obtained an oracle deciding the halting problem (which is expressible in CIC).

Corollary 41. *We have* $\nvdash_{\mathrm{CIC}^p_{\mathbb{E}}}$ MP *and thus also* $\nvdash_{\mathrm{CIC}}$ MP.

5 Possible Extensions

5.1 Negative Records

Interestingly, the fact that the translation introduces effects has unintented consequences on a few properties of type theory that are often taken for granted. Namely, because type theory is pure, there is a widespread confusion amongst type theorists between positive tuples and negative records.

- Positive tuples are defined as a one-constructor inductive type, introduced by this constructor and eliminated by pattern-matching. They do not (and in general cannot, for typing reasons) satisfy definitional η-laws, also known as *surjective pairing.*
- Negative records are defined as a record type, introduced by primitive packing and eliminated by projections. They naturally obey definitional η-laws.

$$A, B, M, N ::= \dots \mid \&x : A.\, B \mid \langle M, N\rangle \mid M.\pi_1 \mid M.\pi_2$$

$$\frac{\Gamma \vdash A : \square_i \quad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \&x : A.\, B : \square_{\max(i,j)}} \qquad \frac{\Gamma \vdash M : \&x : A.\, B}{\Gamma \vdash M.\pi_1 : A} \qquad \frac{\Gamma \vdash M : \&x : A.\, B}{\Gamma \vdash M.\pi_2 : B\{x := M.\pi_1\}}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash B : \square \qquad \Gamma \vdash N : B\{x := M\}}{\Gamma \vdash \langle M, N\rangle : \&x : A.\, B}$$

$$\langle M.\pi_1, M.\pi_2\rangle \equiv M \qquad \langle M, N\rangle.\pi_1 \equiv M \qquad \langle M, N\rangle.\pi_2 \equiv N$$

Fig. 7. Negative pairs

$$\begin{aligned}
[\&x : A.\, B] &:= \texttt{TypeVal}\ (\&x : [\![A]\!].\, [\![B]\!])\ (\lambda e : \mathbb{E}.\, \langle [A]_\varnothing\ e, [B]_\varnothing\{x := [A]_\varnothing\ e\}\ e\rangle) \\
[\langle M, N\rangle] &:= \langle [M], [N]\rangle \\
[M.\pi_i] &:= [M].\pi_i
\end{aligned}$$

Fig. 8. Exceptional translation of negative pairs

In the remainder of this section, we will focus on the specific case of pairs, but the same arguments are generalizable to arbitrary records. Positive pairs $\Sigma x : A.\, B$ are defined by the inductive type from Fig. 4. Negative pairs $\&x : A.\, B$ are defined as a primitive structure in Fig. 7. We use the ampersand notation as a reference to linear logic.

In CIC, it is possible to show that negative and positive pairs are propositionally isomorphic, because positive pairs enjoy dependent elimination. Nonetheless, it is a well-known fact in the programming folklore that in a call-by-name language with effects, the two are sharply distinct. For instance, in presence of exceptions, assuming $\vdash M : \Sigma x : A.\, B$, one does not have in general

$$M \equiv \texttt{ex}\ A\ B\ (\texttt{fst}\ A\ B\ M)\ (\texttt{snd}\ A\ B\ M)$$

where `fst` and `snd` are defined by pattern-matching. Indeed, if M is itself an exception, the two sides can be discriminated by a pattern-matching. Matching on the left-hand side results in immediate reraising of the exception, while matching on the right-hand side succeeds as long as the arguments of the constructor are not forced. Forcefully equating those two terms would then result in a trivial equational theory.

Such a phenomenon is at work in the exceptional translation. It is actually possible to interpret negative pairs through the translation, but in a way that significantly differs from the translation of positive pairs. In this section, we assume that $\mathcal{T}$ contains negative pairs.

Definition 42. *The translation of negative pairs is given in Fig. 8.*

It is straightforward to check that the definitions of Fig. 8 preserve the conversion and typing rules from Fig. 7. The same translation can be extended to any record. We thus have the following theorem.

Theorem 43. *If $\mathcal{T}$ has negative records, then so has $\mathcal{T}_{\mathbb{E}}$.*

It is enlightening to look at the difference between negative and positive pairs through the translation, because now we have effects that allow to separate them clearly. Indeed, compare

$$[\![\&x : A.\, B]\!] \equiv \&x : [\![A]\!].\, [\![B]\!] \quad \text{with} \quad [\![\Sigma x : A.\, B]\!] \cong \mathbb{E} + \Sigma x : [\![A]\!].\, [\![B]\!].$$

Clearly, if $\mathbb{E}$ is inhabited, then the two types do not even have the same cardinal, assuming A and B are finite. Furthermore, their default inhabitant is not the same at all. It is defined pointwise for negative pairs, while it is a special constructor for positive ones. Finally, there is obviously not any chance that $[\![\Sigma x : A.\, B]\!]$ satisfies definitional surjective pairing in vanilla CIC, as it has two constructors. The trick is that the two types are externally distinguishable, but are not internally so, because $\mathcal{T}_{\mathbb{E}}$ is a model of CIC+& and thus proves that they are propositionally isomorphic.

It is possible to equip negative pairs with a parametricity relation defined as a primitive record which is the pointwise parametricity relation of each field, which naturally preserve typing and conversion rules.

Theorem 44. *If $\mathcal{T}$ has negative records, then so has $\mathcal{T}^{p}_{\mathbb{E}}$.*

5.2 Impredicative Universe

All the systems we have considered so far are predicative. It is nonetheless possible to implement an impredicative universe $*$ in $\mathcal{T}_{\mathbb{E}}$ if $\mathcal{T}$ features one.

Intuitively, it is sufficient to ask for an inductive type `prop` living in $\square_i$ for all i, which is defined just as `type`, except that its constructor `PropVal` corresponding to `TypeVal` contains elements of $*$ rather than $\square$. Then one can similarly define $\texttt{El}_*$ and $\texttt{Err}_*$ acting on `prop` rather than `type`. One then slightly tweaks the $[\![\cdot]\!]$ macro from Fig. 2 by defining it instead as

$$[\![A]\!] := \begin{cases} \texttt{El}_*\ [A] \text{ if } A : * \\ \texttt{El}\ [A] \quad \text{otherwise} \end{cases}$$

and similarly for type constructors. With this modified translation, one obtains a soundness theorem for CC_ω.

Theorem 45. *The exceptional translation is a syntactic model of* $\mathrm{CC}_\omega + *$.

Likewise, the inductive translation is amenable to interpret an impredicative universe, with one major restriction though.

Theorem 46. *The exceptional translation is a syntactic model of* CIC $+\, *$ *without the singleton elimination rule.*

Indeed, the addition of the default constructor disrupts the singleton elimination criterion for all inductive types. Actually, this criterion is very fragile, and even if $\mathcal{T}_{\mathbb{E}}$ satisfied it, Keller and Lasson showed that the parametricity translation could not interpret inductive types in $*$ for similar reasons [17], and $\mathcal{T}^p_{\mathbb{E}}$ would face the same issue.

6 The Exceptional Translation in Practice

6.1 Implementation as a Coq Plugin

The (parametric) exceptional translation is a translation of CIC into itself, which means that we can directly implement it as a Coq plugin. This way, we can use the translation to extend safely Coq with new logical principles, so that typechecking remains decidable.

Such a Coq plugin is simply a program that, given a Coq proof term M, produces the translations $[M]$ and $[M]_\varepsilon$ as Coq terms. For instance, the translations of type `list`, given in Figs. 4 and 6, are obtained by typing the following commands, which define each one new inductive type in Coq.

```
Effect Translate list.
Parametricity Translate list.
```

The first command produces only [`list`], while the second produces $[$`list`$]_\varepsilon$. But the main interest of the translation is that we can exhibit new constructors. For instance, the `raise` operation described in Sect. 2.4 is defined as

```
Effect Definition Exception : Type := fun E ⇒ TypeVal E E id.
Effect Definition raise : ∀ A, Exception → A := fun E (A : type E) ⇒ Err A.
```

6.2 Usecase: A Cast Framework

We can use the ability to raise exception to define partial function in the exceptional layer. For instance, given a decidable property (described by the type class below), it is then possible to define a cast function from `A` to `Σ (a : A). P a` returning the converted value if the property is satisfied and raising an exception otherwise (using an inhabitant `cast_failed` of `Exception`).

```
Class Decidable (A : Type) := dec : A + (not A).
Definition cast A (P : A → Type) (a:A) {Hdec : Decidable (P a)} : Σ (a : A). P a
:= match dec (P a) with
   | inl p ⇒ (a ; p)
   | inr _ ⇒ raise cast_failed
   end.
```

Using this cast mechanism, it is easy to define a function `list_to`_pair from lists to pairs by first converting the list into a list size two, using the impure function `cast (list A) (fun l ⇒ List.length l = 2)` and then recovering a pair from a list of size two using a pure function.

In the exceptional layer, it is possible to prove the following property

```
Definition list_to_pair_prop A (x y : A) : list_to_pair [x ; y] = (x,y).
```

in at least two way. One can perfectly prove it by simply raising an exception at top level, or by reflexivity—using the fact that `list_to_pair` [x ; y] actually reduces to (x,y).

However, there is a way to distinguish between those two proofs in the target theory, here Coq, by stating the following lemma which can only proven for the proof not raising an exception.

```
Definition list_to_pair_prop_soundness A x y :
    list_to_pair_prop• A x y = eq_refl• _ _ _ := eq_refl _.
```

where underscores represent arguments inferred by Coq.

7 Related Work

Adding Dependency to an Effectful Language. There are numerous works on adding dependent types in mainstream effectful programming languages. They all mostly focused on how to appropriately restrict effectful terms from appearing in types. Indeed, if types only depend on pure terms, the problem of having two different evaluations of the effect of the term (at the level of types and at the level of terms) disappear. This is the case for instance for Dependent ML of Xi and Pfenning [18], or more recently for Casinghino *et al.* [19] on how to combine proofs and programs when programs can be non-terminating. The $F^{\star}$ programming language of Swamy *et al.* [20] uses a notion of primitive effects including state, exceptions, divergence and IO. Each effect is described through a monadic predicate transformer semantics which allows to have a pure core dependent language to reason on those effects. On a more foundational side, there are two recent and overlapping lines of work on the description of a dependent call-by-push-value (CBPV) by Ahman *et al.* [21] and Vákár [22]. Those works also use a purity restriction for dependency, but using the CBPV language, deals with any effect described in monadic style. On another line of work, Brady advocates for the use of algebraic effects as an elegant way to allow combing effects more smoothly than with a monadic approach and gives an implementation in Idris [23].

Adding Effects to a Dependently-Typed Language. Nanevski *et al.* [24] have developed Hoare type theory (HTT) to extend Coq with monadic style effects. To this end, they provide an axiomatic extension of Coq with a monad in which to encapsulate imperative code. Important tools have been developed on HTT, most notably the Ynot project [25]. Apart from being axiomatic, their monadic approach does not allow to mix effectful programs and dependency but is rather made for proving inside Coq properties on simply typed imperative programs.

Internal Translation of Type Theory. A non-axiomatic way to extend type theory with new features is to use internal translation, that is translation of type theory into itself as advocated by Boulier *et al.* [9]. The presentation of parametricity for type theory given by Bernardy and Lasson [5] can be seen as one of the first internal translation of type theory. However, this one does not add any new power to type theory as it is a conservative extension. Barthe *et al.* [26] have described a CPS translation for CC_ω featuring `call-cc`, but without dealing with inductive types and relying on a form of type stratification. A variant of this translation has been extended recently by Bowman *et al.* [27] to dependent sums using answer-type polymorphism $\Pi\alpha : \Box.\, (A \to \alpha) \to \alpha$. A generic class of internal translations has been defined by Jaber *et al.* [28] using forcing, which can be seen as a type theoretic version of the presheaf construction used in categorical logic. This class of translation works on all CIC but for a restricted version of dependent elimination, identical to the Baclofen type theory [2]. Therefore, to the best of our knowledge, the exceptional translation is the first complete internal translation of CIC adding a particular notion of effect.

8 Conclusion and Future Work

In this paper, we have defined the exceptional translation, the first syntactic translation of the Calculus of Inductive Constructions into itself, adding effects and that covers full dependent elimination. This results in a new type theory, which features call-by-name exceptions with decidable type-checking and a weaker form of canonicity. We have shown that although the resulting theory is inconsistent, it is possible to reason on exceptional programs and show that some of them actually never raise an exception by relying on the target theory. This provides a sound logical framework allowing to transparently prove safety properties about impure dependently-typed programs. Then, using parametricity, we have given an additional layer at the top of the exceptional translation in order to tame exceptions and preserve consistency. This way, we have consistently extended the logical expressivity of CIC with independence of premises, Markov's rule, and the negation of function extensionality while retaining η-expansion. Both translations have been implemented in a Coq plugin, which we use to formalize the examples.

One of the main directions of future work is to investigate whether other kind of effects can give rise to an internal translation of CIC. To that end, it seems promising to look at algebraic presentation of effects. Indeed, the recent work on the non-necessity of the value restriction policy for algebraic effects and handlers of Kammar and Pretnar [29] suggests that we should be able to perform similar translations on CIC with full dependent elimination for other algebraic effects and handlers than exceptions.

Acknowledgements. This research was supported in part by an ERC Consolidator Grant for the project "RustBelt", funded under Horizon 2020 grant agreement № 683289 and an ERC Starting Grant for the project "CoqHoTT", funded under Horizon 2020 grant agreement № 637339.

References

1. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991)
2. Pédrot, P., Tabareau, N.: An effectful way to eliminate addiction to dependence. In: 32nd Annual Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, 20–23 June 2017, pp. 1–12 (2017)
3. Munch-Maccagnoni, G.: Models of a non-associative composition. In: Muscholl, A. (ed.) FoSSaCS 2014. LNCS, vol. 8412, pp. 396–410. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54830-7_26
4. Kreisel, G.: Interpretation of analysis by means of constructive functionals of finite types. In: Heyting, A. (ed.) Constructivity in Mathematics, pp. 101–128. North-Holland Pub. Co., Amsterdam (1959)
5. Bernardy, J.-P., Lasson, M.: Realizability and parametricity in pure type systems. In: Hofmann, M. (ed.) FoSSaCS 2011. LNCS, vol. 6604, pp. 108–122. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19805-2_8
6. Avigad, J., Feferman, S.: Gödel's functional ("Dialectica") interpretation. In: The Handbook of Proof Theory, pp. 337–405. North-Holland (1999)
7. Coquand, T., Mannaa, B.: The independence of Markov's principle in type theory. In: 1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, Porto, Portugal, 22–26 June 2016, pp. 17:1–17:18 (2016)
8. Coquand, T., Huet, G.P.: The calculus of constructions. Inf. Comput. **76**(2/3), 95–120 (1988)
9. Boulier, S., Pédrot, P., Tabareau, N.: The next 700 syntactical models of type theory. In: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, 16–17 January 2017, pp. 182–194 (2017)
10. The Coq Development Team: The Coq proof assistant reference manual (2017)
11. Werner, B.: Une Théorie des Constructions Inductives. Ph.D. thesis, Université Paris-Diderot - Paris VII, May 1994
12. Tanter, É., Tabareau, N.: Gradual certified programming in Coq. In: Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015), Pittsburgh, PA, USA, pp. 26–40. ACM Press, October 2015
13. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress, pp. 513–523 (1983)
14. Friedman, H.: Classically and intuitionistically provably recursive functions. In: Miiller, G.H., Scott, D.S. (eds.) Higher Set Theory. Lecture Notes in Mathematics, pp. 21–27. Springer, Heidelberg (1978)
15. Troelstra, A. (ed.): Metamathematical Investigation of Intuitionistic Arithmetic and Analysis. Lecture Notes in Mathematics, vol. 344. Springer, Heidelberg (1973). https://doi.org/10.1007/BFb0066739
16. Herbelin, H.: An intuitionistic logic that proves Markov's principle. In: Proceedings of the 25th Annual Symposium on Logic in Computer Science, LICS 2010, Edinburgh, United Kingdom, 11–14 July 2010, pp. 50–56 (2010)
17. Keller, C., Lasson, M.: Parametricity in an impredicative sort. In: Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, Fontainebleau, France, 3–6 September 2012, pp. 381–395 (2012)
18. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1999, pp. 214–227. ACM, New York (1999)

19. Casinghino, C., Sjöberg, V., Weirich, S.: Combining proofs and programs in a dependently typed language. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, pp. 33–45. ACM, New York (2014)
20. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 256–270. ACM, January 2016
21. Ahman, D., Ghani, N., Plotkin, G.D.: Dependent types and fibred computational effects. In: Jacobs, B., Löding, C. (eds.) FoSSaCS 2016. LNCS, vol. 9634, pp. 36–54. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49630-5_3
22. Vákár, M.: A framework for dependent types and effects (2015) draft
23. Brady, E.: Idris, a general-purpose dependently typed programming language: design and implementation. J. Funct. Program. **23**(05), 552–593 (2013)
24. Nanevski, A., Morrisett, G., Birkedal, L.: Hoare type theory, polymorphism and separation. J. Funct. Program. **18**(5–6), 865–911 (2008)
25. Chlipala, A., Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Effective interactive proofs for higher-order imperative programs. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP 2009, pp. 79–90. ACM, New York (2009)
26. Barthe, G., Hatcliff, J., Sørensen, M.H.B.: CPS translations and applications: the cube and beyond. High. Order Symbol. Comput. **12**(2), 125–170 (1999)
27. Bowman, W., Cong, Y., Rioux, N., Ahmed, A.: Type-preserving CPS translation of σ and π types is not possible. In: Proceedings of the 45th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2018. ACM, New York (2018)
28. Jaber, G., Lewertowski, G., Pédrot, P., Sozeau, M., Tabareau, N.: The definitional side of the forcing. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2016, New York, NY, USA, 5–8 July 2016, pp. 367–376 (2016)
29. Kammar, O., Pretnar, M.: No value restriction is needed for algebraic effects and handlers. J. Funct. Program. **27**, 367–376 (2017)

Let Arguments Go First

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

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

Abstract. Bi-directional type checking has proved to be an extremely useful and versatile tool for type checking and type inference. The conventional presentation of bi-directional type checking consists of two modes: *inference* mode and *checked* mode. In traditional bi-directional type-checking, type annotations are used to guide (via the checked mode) the type inference/checking procedure to determine the type of an expression, and *type information flows from functions to arguments.*

This paper presents a variant of bi-directional type checking where the *type information flows from arguments to functions.* This variant retains the inference mode, but adds a so-called *application* mode. Such design can remove annotations that basic bi-directional type checking cannot, and is useful when type information from arguments is required to type-check the functions being applied. We present two applications and develop the meta-theory (mostly verified in Coq) of the application mode.

1 Introduction

Bi-directional type checking has been known in the folklore of type systems for a long time. It was popularized by Pierce and Turner's work on *local type inference* [29]. Local type inference was introduced as an alternative to Hindley-Milner (henceforth HM system) type systems [11,17], which could easily deal with polymorphic languages with subtyping. Bi-directional type checking is one component of local type inference that, aided by some type annotations, enables type inference in an expressive language with polymorphism and subtyping. Since Pierce and Turner's work, various other authors have proved the effectiveness of bi-directional type checking in several other settings, including many different systems with subtyping [12,14,15], systems with dependent types [2,3,10,21,37], and various other works [1,7,13,22,28]. Furthermore, bi-directional type checking has also been combined with HM-style techniques for providing type inference in the presence of higher-ranked types [14,27].

The key idea in bi-directional type checking is simple. In its basic form typing is split into *inference* and *checked* modes. The most salient feature of a bi-directional type-checker is when information deduced from inference mode is used to guide checking of an expression in checked mode. One of such interactions between modes happens in the typing rule for function applications:

$$\frac{\Gamma \vdash e_1 \ \Rightarrow\ A \to B \qquad \Gamma \vdash e_2 \ \Leftarrow\ A}{\Gamma \vdash e_1\ e_2 \ \Rightarrow\ B}\ \text{APP}$$

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 272–299, 2018.
https://doi.org/10.1007/978-3-319-89884-1_10

In the above rule, which is a standard bi-directional rule for checking applications, the two modes are used. First we synthesize ($\Rightarrow$) the type $A \rightarrow B$ from e_1, and then check ($\Leftarrow$) e_2 against A, returning B as the type for the application.

This paper presents a variant of bi-directional type checking that employs a so-called *application* mode. With the application mode the design of the application rule (for a simply typed calculus) is as follows:

$$\frac{\Gamma \vdash e_2 \ \Rightarrow \ A \qquad \Gamma \mid \Psi, A \vdash e_1 \ \Rightarrow \ A \rightarrow B}{\Gamma \mid \Psi \vdash e_1 \ e_2 \ \Rightarrow \ B}\ \text{APP}$$

In this rule, there are two kinds of judgments. The first judgment is just the usual inference mode, which is used to infer the type of the argument e_2. The second judgment, the application mode, is similar to the inference mode, but it has an additional context Ψ. The context Ψ is a stack that tracks the types of the arguments of outer applications. In the rule for application, the type of the argument e_2 is inferred first, and then pushed into Ψ for inferring the type of e_1. Applications are themselves in the application mode, since they can be in the context of an outer application. With the application mode it is possible to infer the type for expressions such as `(λx. x) 1` without additional annotations.

Bi-directional type checking with an application mode may still require type annotations and it gives different trade-offs with respect to the checked mode in terms of type annotations. However the different trade-offs open paths to different designs of type checking/inference algorithms. To illustrate the utility of the application mode, we present two different calculi as applications. The first calculus is a higher ranked implicit polymorphic type system, which infers higher-ranked types, generalizes the HM type system, and has polymorphic **let** as syntactic sugar. As far as we are aware, no previous work enables an HM-style let construct to be expressed as syntactic sugar. For this calculus many results are proved using the Coq proof assistant [9], including type-safety. Moreover a sound and complete algorithmic system, inspired by Peyton Jones et al. [27], is also developed. A second calculus with *explicit polymorphism* illustrates how the application mode is compatible with type applications, and how it adds expressiveness by enabling an encoding of type declarations in a System-F-like calculus. For this calculus, all proofs (including type soundness), are mechanized in Coq.

We believe that, similarly to standard bi-directional type checking, bi-directional type checking with an application mode can be applied to a wide range of type systems. Our work shows two particular and non-trivial applications. Other potential areas of applications are other type systems with subtyping, static overloading, implicit parameters or dependent types.

In summary the contributions of this paper are[1]:

- **A variant of bi-directional type checking** where the inference mode is combined with a new, so-called, application mode. The application mode naturally propagates type information from arguments to the functions.
- **A new design for type inference of higher-ranked types** which generalizes the HM type system, supports a polymorphic **let** as syntactic sugar, and infers higher rank types. We present a syntax-directed specification, an elaboration semantics to System F, some meta-theory in Coq, and an algorithmic type system with completeness and soundness proofs.
- **A System-F-like calculus** as a theoretical response to the challenge noted by Pierce and Turner [29]. It shows that the application mode is compatible with type applications, which also enables encoding type declarations. We present a type system and meta-theory, including proofs of type safety and uniqueness of typing in Coq.

2 Overview

2.1 Background: Bi-directional Type Checking

Traditional type checking rules can be heavyweight on annotations, in the sense that lambda-bound variables always need explicit annotations. Bi-directional type checking [29] provides an alternative, which allows types to propagate downward the syntax tree. For example, in the expression `(λf:Int → Int. f) (λy. y)`, the type of y is provided by the type annotation on `f`. This is supported by the bi-directional typing rule for applications:

$$\frac{\Gamma \vdash e_1 \ \Rightarrow\ A \to B \qquad \Gamma \vdash e_2 \ \Leftarrow\ A}{\Gamma \vdash e_1\ e_2 \ \Rightarrow\ B}\ \text{APP}$$

Specifically, if we know that the type of e_1 is a function from `A → B`, we can check that e_2 has type `A`. Notice that here the type information flows from functions to arguments.

One guideline for designing bi-directional type checking rules [15] is to distinguish introduction rules from elimination rules. Constructs which correspond to introduction forms are *checked* against a given type, while constructs corresponding to elimination forms *infer* (or synthesize) their types. For instance, under this design principle, the introduction rule for pairs is supposed to be in checked mode, as in the rule Pair-C.

$$\frac{\Gamma \vdash e_1 \ \Leftarrow\ A \qquad \Gamma \vdash e_2 \ \Leftarrow\ B}{\Gamma \vdash (e_1, e_2) \ \Leftarrow\ (A, B)}\ \textsc{Pair-C} \qquad \frac{\Gamma \vdash e_1 \ \Rightarrow\ A \qquad \Gamma \vdash e_2 \ \Rightarrow\ B}{\Gamma \vdash (e_1, e_2) \ \Rightarrow\ (A, B)}\ \textsc{Pair-I}$$

[1] All supplementary materials are available in https://bitbucket.org/ningningxie/let-arguments-go-first.

Unfortunately, this means that the trivial program `(1, 2)` cannot type-check, which in this case has to be rewritten to `(1, 2) : (Int , Int)`.

In this particular case, bi-directional type checking goes against its original intention of removing burden from programmers, since a seemingly unnecessary annotation is needed. Therefore, in practice, bi-directional type systems do not strictly follow the guideline, and usually have additional inference rules for the introduction form of constructs. For pairs, the corresponding rule is Pair-I.

Now we can type check `(1, 2)`, but the price to pay is that two typing rules for pairs are needed. Worse still, the same criticism applies to other constructs. This shows one drawback of bi-directional type checking: often to minimize annotations, many rules are duplicated for having both inference and checked mode, which scales up with the typing rules in a type system.

2.2 Bi-directional Type Checking with the Application Mode

We propose a variant of bi-directional type checking with a new *application mode*. The application mode preserves the advantage of bi-directional type checking, namely many redundant annotations are removed, while certain programs can type check with even fewer annotations. Also, with our proposal, the inference mode is a special case of the application mode, so it does not produce duplications of rules in the type system. Additionally, the checked mode can still be *easily* combined into the system (see Sect. 5.1 for details). The essential idea of the application mode is to enable the type information flow in applications to propagate from arguments to functions (instead of from functions to arguments as in traditional bi-directional type checking).

To motivate the design of bi-directional type checking with an application mode, consider the simple expression

```
(λx. x) 1
```

This expression cannot type check in traditional bi-directional type checking because unannotated abstractions only have a checked mode, so annotations are required. For example, `((λx. x) : Int → Int) 1`.

In this example we can observe that if the type of the argument is accounted for in inferring the type of `λx. x`, then it is actually possible to deduce that the lambda expression has type `Int → Int` , from the argument `1`.

The Application Mode. If types flow from the arguments to the function, an alternative idea is to push the type of the arguments into the typing of the function, as the rule that is briefly introduced in Sect. 1:

$$\frac{\Gamma \vdash e_2 \;\Rightarrow\; A \qquad \Gamma \shortmid \Psi, A \vdash e_1 \;\Rightarrow\; A \to B}{\Gamma \shortmid \Psi \vdash e_1\ e_2 \;\Rightarrow\; B}\ \text{APP}$$

Here the argument e_2 synthesizes its type `A`, which then is pushed into the application context Ψ. Lambda expressions can now make use of the application context, leading to the following rule:

$$\frac{\Gamma, x : A \mid \Psi \vdash e \Rightarrow B}{\Gamma \mid \Psi, A \vdash \lambda x.\, e \Rightarrow A \to B}\ \textsc{Lam}$$

The type `A` that appears last in the application context serves as the type for `x`, and type checking continues with a smaller application context and `x:A` in the typing context. Therefore, using the rule App and Lam, the expression `(λx. x) 1` can type-check without annotations, since the type `Int` of the argument `1` is used as the type of the binding `x`.

Note that, since the examples so far are based on simple types, obviously they can be solved by integrating type inference and relying on techniques like unification or constraint solving. However, here the point is that the application mode helps to reduce the number of annotations *without requiring such sophisticated techniques*. Also, the application mode helps with situations where those techniques cannot be easily applied, such as type systems with subtyping.

Interpretation of the Application Mode. As we have seen, the guideline for designing bi-directional type checking [15], based on introduction and elimination rules, is often not enough in practice. This leads to extra introduction rules in the inference mode. The application mode does not distinguish between introduction rules and elimination rules. Instead, to decide whether a rule should be in inference or application mode, we need to think whether the expression can be applied or not. Variables, lambda expressions and applications are all examples of expressions that can be applied, and they should have application mode rules. However pairs or literals cannot be applied and should have inference rules. For example, type checking pairs would simply lead to the rule Pair-I. Nevertheless elimination rules of pairs could have non-empty application contexts (see Sect. 5.2 for details). In the application mode, arguments are always inferred first in applications and propagated through application contexts. An empty application context means that an expression is not being applied to anything, which allows us to model the inference mode as a particular case[2].

Partial Type Checking. The inference mode synthesizes the type of an expression, and the checked mode checks an expression against some type. A natural question is how do these modes compare to application mode. An answer is that, in some sense: the application mode is stronger than inference mode, but weaker than checked mode. Specifically, the inference mode means that we know nothing about the type an expression before hand. The checked mode means that the whole type of the expression is already known before hand. With the application mode we know some partial type information about the type of an expression:

[2] Although the application mode generalizes the inference mode, we refer to them as two different modes. Thus the variant of bi-directional type checking in this paper is interpreted as a type system with both *inference* and *application* modes.

we know some of its argument types (since it must be a function type when the application context is non-empty), but not the return type.

Instead of nothing or all, this partialness gives us a finer grain notion on how much we know about the type of an expression. For example, assume $e : A \to B \to C$. In the inference mode, we only have e. In the checked mode, we have both e and $A \to B \to C$. In the application mode, we have e, and maybe an empty context (which degenerates into inference mode), or an application context A (we know the type of first argument), or an application context B, A (we know the types of both arguments).

Trade-offs. Note that the application mode is *not* conservative over traditional bidirectional type checking due to the different information flow. However, it provides a new design choice for type inference/checking algorithms, especially for those where the information about arguments is useful. Therefore we next discuss some benefits of the application mode for two interesting cases where functions are either variables; or lambda (or type) abstractions.

2.3 Benefits of Information Flowing from Arguments to Functions

Local Constraint Solver for Function Variables. Many type systems, including type systems with *implicit polymorphism* and/or *static overloading*, need information about the types of the arguments when type checking function variables. For example, in conventional functional languages with implicit polymorphism, function calls such as `(id 3)` where `id: ∀a. (a → a)`, are *pervasive*. In such a function call the type system must instantiate a to Int. Dealing with such implicit instantiation gets trickier in systems with *higher-ranked types*. For example, Peyton Jones et al. [27] require additional syntactic forms and relations, whereas Dunfield and Krishnaswami [14] add a special purpose *application judgment*.

With the application mode, all the type information about the arguments being applied is available in application contexts and can be used to solve instantiation constraints. To exploit such information, the type system employs a special subtyping judgment called *application subtyping*, with the form $\Psi \vdash A \le B$. Unlike conventional subtyping, computationally Ψ and A are interpreted as inputs and B as output. In above example, we have that $\mathsf{Int} \vdash \forall a.a \to a \le B$ and we can determine that $a = \mathsf{Int}$ and $B = \mathsf{Int} \to \mathsf{Int}$. In this way, type system is able to solve the constraints *locally* according to the application contexts since we no longer need to propagate the instantiation constraints to the typing process.

Declaration Desugaring for Lambda Abstractions. An interesting consequence of the usage of an application mode is that it enables the following **let** sugar:

$$\textbf{let}\ \mathtt{x}\ \mathtt{=}\ e_1\ \textbf{in}\ e_2 \rightsquigarrow (\lambda\mathtt{x}.\ e_2)\ e_1$$

Such syntactic sugar for **let** is, of course, standard. However, in the context of implementations of typed languages it normally requires extra type annotations or a more sophisticated type-directed translation. Type checking $(\lambda x.\ e_2)\ e_1$

would normally require annotations (for example an annotation for x), or otherwise such annotation should be inferred first. Nevertheless, with the application mode no extra annotations/inference is required, since from the type of the argument e_1 it is possible to deduce the type of x. Generally speaking, with the application mode *annotations are never needed for applied lambdas*. Thus **let** can be the usual sugar from the untyped lambda calculus, including HM-style **let** expression and even type declarations.

2.4 Application 1: Type Inference of Higher-Ranked Types

As a first illustration of the utility of the application mode, we present a calculus with *implicit predicative higher-ranked polymorphism*.

Higher-Ranked Types. Type systems with higher-ranked types generalize the traditional HM type system, and are useful in practice in languages like Haskell or other ML-like languages. Essentially higher-ranked types enable much of the expressive power of System F, with the advantage of implicit polymorphism. Complete type inference for System F is known to be undecidable [36]. Therefore, several partial type inference algorithms, exploiting additional type annotations, have been proposed in the past instead [15,25,27,31].

Higher-Ranked Types and Bi-directional Type Checking. Bi-directional type checking is also used to help with the inference of higher-ranked types [14,27]. Consider the following program:

```
(λf. (f 1, f 'c')) (λx. x)
```

which is not typeable under those type systems because they fail to infer the type of f, since it is supposed to be polymorphic. Using bi-directional type checking, we can rewrite this program as

```
((λf. (f 1, f 'c')) : (∀a. a → a) → (Int, Char)) (λx . x)
```

Here the type of f can be easily derived from the type signature using checked mode in bi-directional type checking. However, although some redundant annotations are removed by bi-directional type checking, the burden of inferring higher-ranked types is still carried by programmers: they are forced to add polymorphic annotations to help with the type derivation of higher-ranked types. For the above example, the type annotation is still *provided by programmers*, even though the necessary type information can be derived intuitively without any annotations: f is applied to `λx. x`, which is of type `∀a. a → a`.

Generalization. Generalization is famous for its application in let polymorphism in the HM system, where generalization is adopted at let bindings. Let polymorphism is a useful component to introduce top-level quantifiers (rank 1 types) into a polymorphic type system. The previous example becomes typeable in the HM system if we rewrite it to: **let** `f = λx. x` **in** `(f 1, f 'c')`.

Type Inference for Higher-Ranked Types with the Application Mode. Using our bi-directional type system with an application mode, the original expression can type check without annotations or rewrites: `(λf. (f 1, f 'c')) (λx. x)`.

This result comes naturally if we allow type information flow from arguments to functions. For inferring polymorphic types for arguments, we use *generalization*. In the above example, we first infer the type `∀a. a → a` for the argument, then pass the type to the function. A nice consequence of such an approach is that HM-style polymorphic **let** expressions are simply regarded as syntactic sugar to a combination of lambda/application:

let `x =` e_1 **in** $e_2 \rightsquigarrow$ `(λx.` e_2`)` e_1

With this approach, nested lets can lead to types which are *more general* than HM. For example, **let** `s = λx. x` **in let** `t = λy. s` **in** `e`. The type of `s` is `∀a. a → a` after generalization. Because `t` returns `s` as a result, we might expect `t: ∀b. b → (∀a. a → a)`, which is what our system will return. However, HM will return type `t: ∀b. ∀a. b → (a → a)`, as it can only return rank 1 types, which is less general than the previous one according to Odersky and Läufer's subtyping relation for polymorphic types [24].

Conservativity over the Hindley-Milner Type System. Our type system is a conservative extension over the Hindley-Milner type system, in the sense that every program that can type-check in HM is accepted in our type system, which is explained in detail in Sect. 3.2. This result is not surprising: after desugaring **let** into a lambda and an application, programs remain typeable.

Comparing Predicative Higher-Ranked Type Inference Systems. We will give a full discussion and comparison of related work in Sect. 6. Among those works, we believe the work by Dunfield and Krishnaswami [14], and the work by Peyton Jones et al. [27] are the most closely related work to our system. Both their systems and ours are based on a *predicative* type system: universal quantifiers can only be instantiated by monotypes. So we would like to emphasize our system's properties in relation to those works. In particular, here we discuss two interesting differences, and also briefly (and informally) discuss how the works compare in terms of expressiveness.

(1) Inference of higher-ranked types. In both works, every polymorphic type inferred by the system must correspond to one annotation provided by the programmer. However, in our system, some higher-ranked types can be inferred from the expression itself without any annotation. The motivating expression above provides an example of this.

(2) Where are annotations needed? Since type annotations are useful for inferring higher rank types, a clear answer to the question where annotations are needed is necessary so that programmers know when they are required to write annotations. To this question, previous systems give a concrete answer: only on the binding of polymorphic types. Our answer is slightly different: only on the bindings of polymorphic types in abstractions *that are not applied to arguments*. Roughly speaking this means that our system ends up with fewer or smaller annotations.

(3) Expressiveness. Based on these two answers, it may seem that our system should accept all expressions that are typeable in their system. However, this is not true because the application mode is *not* conservative over traditional bi-directional type checking. Consider the expression `(λf : (∀a. a → a) → (Int, Char). f) (λg. (g 1, g 'a'))`, which is typeable in their system. In this case, even if g is a polymorphic binding without a type annotation the expression can still type-check. This is because the original application rule propagates the information from the outer binding into the inner expressions. Note that the fact that such expression type-checks does not contradict their guideline of providing type annotations for every polymorphic binder. Programmers that strictly follow their guideline can still add a polymorphic type annotation for g. However it does mean that it is a little harder to understand where annotations for polymorphic binders can be *omitted* in their system. This requires understanding how the applications in checked mode operate.

In our system the above expression is not typeable, as a consequence of the information flow in the application mode. However, following our guideline for annotations leads to a program that can be type-checked with a smaller annotation: `(λf. f) (λg : (∀a. a → a). (g 1, g 'a'))`. This means that our work is not conservative over their work, which is due to the design choice of the application typing rule. Nevertheless, we can always rewrite programs using our guideline, which often leads to fewer/smaller annotations.

2.5 Application 2: More Expressive Type Applications

The design choice of propagating arguments to functions was subject to consideration in the original work on local type inference [29], but was rejected due to possible non-determinism introduced by explicit type applications:

"It is possible, of course, to come up with examples where it would be beneficial to synthesize the argument types first and then use the resulting information to avoid type annotations in the function part of an application expression.... Unfortunately this refinement does not help infer the type of polymorphic functions. For example, we cannot uniquely determine the type of x in the expression $(fun[X](x)\ e)$ [Int] 3." [29]

Therefore, as a response to this challenge, our second application is a variant of System F. Our development of the calculus shows that the application mode can actually work well with calculi with explicit type applications. To explain the new design, consider the expression:

```
(Λa. λx : a. x + 1) Int
```

which is not typeable in the traditional type system for System F. In System F the lambda abstractions do not account for the context of possible function applications. Therefore when type checking the inner body of the lambda abstraction, the expression `x + 1` is ill-typed, because all that is known is that `x` has the (abstract) type `a`.

If we are allowed to propagate type information from arguments to functions, then we can verify that `a = Int` and `x + 1` is well-typed. The key insight in the new type system is to use application contexts to track type equalities induced by type applications. This enables us to type check expressions such as the body of the lambda above (`x + 1`). Therefore, back to the problematic expression $(fun[X](x)\ e)$ [Int] 3, the type of `x` can be inferred as either `X` or `Int` since they are actually equivalent.

Sugar for Type Synonyms. In the same way that we can regard **let** expressions as syntactic sugar, in the new type system we further *gain built-in type synonyms for free.* A *type synonym* is a new name for an existing type. Type synonyms are common in languages such as Haskell. In our calculus a simple form of type synonyms can be desugared as follows:

```
type a = A in e ⇝ (Λa. e) A
```

One practical benefit of such syntactic sugar is that it enables a direct encoding of a System F-like language with declarations (including type-synonyms). Although declarations are often viewed as a routine extension to a calculus, and are not formally studied, they are highly relevant in practice. Therefore, a more realistic formalization of a programming language should directly account for declarations. By providing a way to encode declarations, our new calculus enables a simple way to formalize declarations.

Type Abstraction. The type equalities introduced by type applications may seem like we are breaking System F type abstraction. However, we argue that *type abstraction* is still supported by our System F variant. For example:

```
let inc = Λa. λx : a. x + 1 in inc Int e
```

(after desugaring) does *not* type-check, as in a System-F like language. In our type system lambda abstractions that are immediatelly applied to an argument, and unapplied lambda abstractions behave differently. Unapplied lambda abstractions are just like System F abstractions and retain type abstraction. The example above illustrates this. In contrast the typeable example `(Λa. λx : a. x + 1) Int`, which uses a lambda abstraction directly applied to an argument, can be regarded as the desugared expression for **type** `a = Int` **in** `λx : a . x + 1`.

3 A Polymorphic Language with Higher-Ranked Types

This section first presents a declarative, *syntax-directed* type system for a lambda calculus with implicit higher-ranked polymorphism. The interesting aspects about the new type system are: (1) the typing rules, which employ a combination of inference and application modes; (2) the novel subtyping relation under an application context. Later, we prove our type system is type-safe by a type directed translation to System F [16,27] in Sect. 3.4. Finally an algorithmic type system is discussed in Sect. 3.5.

3.1 Syntax

The syntax of the language is:

$$\begin{array}{lrl} \text{Expr} & e & ::= x \mid n \mid \lambda x : A.\ e \mid \lambda x.\ e \mid e_1\ e_2 \\ \text{Type} & A, B & ::= a \mid A \to B \mid \forall a.A \mid \mathsf{Int} \\ \text{Monotype} & \tau & ::= a \mid \tau_1 \to \tau_2 \mid \mathsf{Int} \\ \text{Typing Context} & \Gamma & ::= \varnothing \mid \Gamma, x : A \\ \text{Application Context} & \Psi & ::= \varnothing \mid \Psi, A \end{array}$$

Expressions. Expressions e include variables (x), integers (n), annotated lambda abstractions ($\lambda x : A.\ e$), lambda abstractions ($\lambda x.\ e$), and applications ($e_1\ e_2$). Letters x, y, z are used to denote term variables. Notably, the syntax does not include a **let** expression ($\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$). Let expressions can be regarded as the standard syntax sugar $(\lambda x.\ e_2)\ e_1$, as illustrated in more detail later.

Types. Types include type variables (a), functions ($A \to B$), polymorphic types ($\forall a.A$) and integers (Int). We use capital letters (A, B) for types, and small letters (a, b) for type variables. Monotypes are types without universal quantifiers.

Contexts. Typing contexts Γ are standard: they map a term variable x to its type A. We implicitly assume that all the variables in Γ are distinct. The main novelty lies in the *application contexts* Ψ, which are the main data structure needed to allow types to flow from arguments to functions. Application contexts are modeled as a stack. The stack collects the types of arguments in applications. The context is a stack because if a type is pushed last then it will be popped first. For example, inferring expression e under application context (a, Int), means e is now being applied to two arguments e_1, e_2, with $e_1 : \mathsf{Int}$, $e_2 : a$, so e should be of type $\mathsf{Int} \to a \to A$ for some A.

3.2 Type System

The top part of Fig. 1 gives the typing rules for our language. The judgment $\Gamma \mid \Psi \vdash e \Rightarrow B$ is read as: under typing context Γ, and application context Ψ, e has type B. The standard inference mode $\Gamma \vdash e \Rightarrow B$ can be regarded as a special case when the application context is empty. Note that the variable names are assumed to be fresh enough when new variables are added into the typing context, or when generating new type variables.

Rule T-Var says that if $x : A$ is in the typing context, and A is a subtype of B under application context Ψ, then x has type B. It depends on the subtyping rules that are explained in Sect. 3.3. Rule T-Int shows that integer literals are only inferred to have type Int under an empty application context. This is obvious since an integer cannot accept any arguments.

T-Lam shows the strength of application contexts. It states that, without annotations, if the application context is non-empty, a type can be popped from the application context to serve as the type for x. Inference of the body then continues with the rest of the application context. This is possible, because the

$$\boxed{\Gamma \mid \Psi \vdash e \;\Rightarrow\; B}$$

$$\frac{x : A \in \Gamma \qquad \Psi \vdash A \;<:\; B}{\Gamma \mid \Psi \vdash x \;\Rightarrow\; B}\;\textsc{T-Var} \qquad \frac{}{\Gamma \vdash n \;\Rightarrow\; \mathsf{Int}}\;\textsc{T-Int}$$

$$\frac{\Gamma, x : A \mid \Psi \vdash e \;\Rightarrow\; B}{\Gamma \mid \Psi, A \vdash \lambda x.\ e \;\Rightarrow\; A \to B}\;\textsc{T-Lam} \qquad \frac{\Gamma, x : \tau \vdash e \;\Rightarrow\; B}{\Gamma \vdash \lambda x.\ e \;\Rightarrow\; \tau \to B}\;\textsc{T-Lam2}$$

$$\frac{\Gamma, x : A \vdash e \;\Rightarrow\; B}{\Gamma \vdash \lambda x : A.\ e \;\Rightarrow\; A \to B}\;\textsc{T-LamAnn1}$$

$$\frac{C \;<:\; A \qquad \Gamma, x : A \mid \Psi \vdash e \;\Rightarrow\; B}{\Gamma \mid \Psi, C \vdash \lambda x : A.\ e \;\Rightarrow\; C \to B}\;\textsc{T-LamAnn2} \qquad \frac{\bar{a} = ftv(A) - ftv(\Gamma)}{\Gamma_{gen}(A) = \forall \bar{a}.A}\;\textsc{T-Gen}$$

$$\frac{\Gamma \vdash e_2 \;\Rightarrow\; A \qquad \Gamma_{gen}(A) = B \qquad \Gamma \mid \Psi, B \vdash e_1 \;\Rightarrow\; B \to C}{\Gamma \mid \Psi \vdash e_1\ e_2 \;\Rightarrow\; C}\;\textsc{T-App}$$

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

$$\frac{}{\mathsf{Int} \;<:\; \mathsf{Int}}\;\textsc{S-Int} \qquad \frac{}{a \;<:\; a}\;\textsc{S-Var} \qquad \frac{A \;<:\; B}{A \;<:\; \forall a.B}\;\textsc{S-ForallR}$$

$$\frac{A[\![a \mapsto \tau]\!] \;<:\; B}{\forall a.A \;<:\; B}\;\textsc{S-ForallL} \qquad \frac{C \;<:\; A \qquad B \;<:\; D}{A \to B \;<:\; C \to D}\;\textsc{S-Fun}$$

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

$$\frac{}{\varnothing \vdash A \;<:\; A}\;\textsc{S-Empty} \qquad \frac{\Psi, C \vdash A[\![a \mapsto \tau]\!] \;<:\; B}{\Psi, C \vdash \forall a.A \;<:\; B}\;\textsc{S-ForallL2}$$

$$\frac{C \;<:\; A \qquad \Psi \vdash B \;<:\; D}{\Psi, C \vdash A \to B \;<:\; C \to D}\;\textsc{S-Fun2}$$

Fig. 1. Syntax-directed typing and subtyping.

expression $\lambda x.\ e$ is being applied to an argument of type A, which is the type at the top of the application context stack. Rule T-Lam2 deals with the case when the application context is empty. In this situation, a monotype τ is *guessed* for the argument, just like the Hindley-Milner system.

Rule T-LamAnn1 works as expected with an empty application context: a new variable x is put with its type A into the typing context, and inference continues on the abstraction body. If the application context is non-empty, then the rule T-LamAnn2 applies. It checks that C is a subtype of A before putting $x : A$ in the typing context. However, note that it is always possible to remove annotations in an abstraction if it has been applied to some arguments.

Rule T-App pushes types into the application context. The application rule first infers the type of the argument e_2 with type A. Then the type A is generalized in the same way that types in **let** expressions are generalized in the HM type system. The resulting generalized type is B. The generalization is shown in rule T-Gen, where all free type variables are extracted to quantifiers. Thus the type of e_1 is now inferred under an application context extended with type B. The generalization step is important to infer higher ranked types: since B is a possibly polymorphic type, which is the argument type of e_1, then e_1 is of possibly a higher rank type.

***Let** Expressions.* The language does not have built-in **let** expressions, but instead supports **let** as syntactic sugar. The typing rule for **let** expressions in the HM system is (without the gray-shaded part):

$$\frac{\Gamma \vdash e_1 \Rightarrow A_1 \qquad \Gamma_{gen}(A_1) = A_2 \qquad \Gamma, x : A_2 \mid \Psi \vdash e_2 \Rightarrow B}{\Gamma \mid \Psi \vdash \mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2 \Rightarrow B}\ \text{T-Let}$$

where we do generalization on the type of e_1, which is then assigned as the type of x while inferring e_2. Adapting this rule to our system with application contexts would result in the gray-shaded part, where the application context is only used for e_2, because e_2 is the expression being applied. If we desugar the **let** expression ($\mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2$) to $((\lambda x.\ e_2)\ e_1)$, we have the following derivation:

$$\frac{\Gamma \vdash e_1 \Rightarrow A_1 \qquad \Gamma_{gen}(A_1) = A_2 \qquad \dfrac{\Gamma, x : A_2 \mid \Psi \vdash e_2 \Rightarrow B}{\Gamma \mid \Psi, A_2 \vdash \lambda x.\ e_2 \Rightarrow A_2 \to B}\ \text{T-Lam}}{\Gamma \mid \Psi \vdash (\lambda x.\ e_2)\ e_1 \Rightarrow B}\ \text{T-App}$$

The type A_2 is now pushed into application context in rule T-App, and then assigned to x in T-Lam. Comparing this with the typing derivations with rule T-Let, we now have same preconditions. Thus we can see that the rules in Fig. 1 are sufficient to express an HM-style polymorphic let construct.

Meta-Theory. The type system enjoys several interesting properties, especially lemmas about application contexts. Before we present those lemmas, we need a helper definition of what it means to use arrows on application contexts.

Definition 1 ($\Psi \to B$). If $\Psi = A_1, A_2, ..., A_n$, then $\Psi \to B$ means the function type $A_n \to ... \to A_2 \to A_1 \to B$.

Such definition is useful to reason about the typing result with application contexts. One specific property is that the application context determines the form of the typing result.

Lemma 1 (Ψ Coincides with Typing Results). *If $\Gamma \mid \Psi \vdash e \Rightarrow A$, then for some A', we have $A = \Psi \to A'$.*

Having this lemma, we can always use the judgment $\Gamma \mid \Psi \vdash e \Rightarrow \Psi \to A'$ instead of $\Gamma \mid \Psi \vdash e \Rightarrow A$.

In traditional bi-directional type checking, we often have one subsumption rule that transfers between inference and checked mode, which states that if an expression can be inferred to some type, then it can be checked with this type. In our system, we regard the normal inference mode $\Gamma \vdash e \Rightarrow A$ as a special case, when the application context is empty. We can also turn from normal inference mode into application mode with an application context.

Lemma 2 (Subsumption). *If $\Gamma \vdash e \Rightarrow \Psi \to A$, then $\Gamma \mid \Psi \vdash e \Rightarrow \Psi \to A$.*

The relationship between our system and standard Hindley Milner type system can be established through the desugaring of let expressions. Namely, if e is typeable in Hindley Milner system, then the desugared expression $|e|$ is typeable in our system, with a more general typing result.

Lemma 3 (Conservative over HM). *If $\Gamma \vdash^{HM} e \Rightarrow A$, then for some B, we have $\Gamma \vdash |e| \Rightarrow B$, and $B <: A$.*

3.3 Subtyping

We present our subtyping rules at the bottom of Fig. 1. Interestingly, our subtyping has two different forms.

Subtyping. The first judgment follows Odersky and Läufer [24]. $A <: B$ means that A is more polymorphic than B and, equivalently, A is a subtype of B. Rules S-Int and S-Var are trivial. Rule S-ForallR states A is subtype of $\forall a.B$ only if A is a subtype of B, with the assumption a is a fresh variable. Rule S-ForallL says $\forall a.A$ is a subtype of B if we can instantiate it with some τ and show the result is a subtype of B. In rule S-Fun, we see that subtyping is contra-variant on the argument type, and covariant on the return type.

Application Subtyping. The typing rule T-Var uses the second subtyping judgment $\Psi \vdash A <: B$. To motivate this new kind of judgment, consider the expression `id 1` for example, whose derivation is stuck at T-Var (here we assume $\mathsf{id} : \forall a.a \to a \in \Gamma$):

$$\dfrac{\Gamma \vdash 1 \Rightarrow \mathsf{Int} \qquad \Gamma_{gen}(\mathsf{Int}) = \mathsf{Int} \qquad \dfrac{\mathsf{id} : \forall a.a \to a \in \Gamma \qquad ???}{\Gamma \mid \mathsf{Int} \vdash \mathsf{id} \Rightarrow}\ \textsc{T-Var}}{\Gamma \vdash \mathsf{id}\ 1 \Rightarrow}\ \textsc{T-App}$$

Here we know that $\mathsf{id} : \forall a.a \to a$ and also, from the application context, that id is applied to an argument of type Int. Thus we need a mechanism for solving the instantiation $a = \mathsf{Int}$ and return a supertype $\mathsf{Int} \to \mathsf{Int}$ as the type of id. This is precisely what the application subtyping achieves: resolve instantiation constraints according to the application context. Notice that unlike existing works [14,27], application subtyping provides a way to solve instantiation more *locally*, since it does not mutually depend on typing.

Back to the rules in Fig. 1, one way to understand the judgment $\Psi \vdash A <: B$ from a computational point-of-view is that the type B is a *computed* output, rather than an input. In other words B is determined from Ψ and A. This is unlike the judgment $A <: B$, where both A and B would be computationally interpreted as inputs. Therefore it is not possible to view $A <: B$ as a special case of $\Psi \vdash A <: B$ where Ψ is empty.

There are three rules dealing with application contexts. Rule S-Empty is for case when the application context is empty. Because it is empty, we have no constraints on the type, so we return it back unchanged. Note that this is where HM systems (also Peyton Jones et al. [27]) would normally use a rule Inst to remove top-level quantifiers:

$$\frac{}{\forall \bar{a}.A <: A[\![\bar{a} \mapsto \bar{\tau}]\!]}\ \textsc{Inst}$$

Our system does not need Inst, because in applications, type information flows from arguments to the function, instead of function to arguments. In the latter case, Inst is needed because a function type is wanted instead of a polymorphic type. In our approach, instantiation of type variables is avoided unless necessary.

The two remaining rules apply when the application context is non-empty, for polymorphic and function types respectively. Note that we only need to deal with these two cases because Int or type variables a cannot have a non-empty application context. In rule S-Forall2, we instantiate the polymorphic type with some τ, and continue. This instantiation is forced by the application context. In rule S-Fun2, one function of type $A \to B$ is now being applied to an argument of type C. So we check $C <: A$. Then we continue with B and the rest application context, and return $C \to D$ as the result type of the function.

Meta-Theory. Application subtyping is novel in our system, and it enjoys some interesting properties. For example, similarly to typing, the application context decides the form of the supertype.

Lemma 4 (Ψ Coincides with Subtyping Results). *If $\Psi \vdash A <: B$, then for some B', $B = \Psi \to B'$.*

Therefore we can always use the judgment $\Psi \vdash A <: \Psi \to B'$, instead of $\Psi \vdash A <: B$. Application subtyping is also reflexive and transitive. Interestingly, in those lemmas, if we remove all applications contexts, they are exactly the reflexivity and transitivity of traditional subtyping.

Lemma 5 (Reflexivity). $\Psi \vdash \Psi \to A <: \Psi \to A$.

Lemma 6 (Transitivity). *If $\Psi_1 \vdash A <: \Psi_1 \to B$, and $\Psi_2 \vdash B <: \Psi_2 \to C$, then $\Psi_2, \Psi_1 \vdash A <: \Psi_1 \to \Psi_2 \to C$.*

Finally, we can convert between subtyping and application subtyping. We can remove the application context and still get a subtyping relation:

Lemma 7 ($\Psi \vdash <:$ to $<:$). *If $\Psi \vdash A <: B$, then $A <: B$.*

Transferring from subtyping to application subtyping will result in a more general type.

Lemma 8 **($<:$ to $\Psi \vdash <:$).** *If* $A <: \Psi \to B_1$*, then for some* B_2*, we have* $\Psi \vdash A <: \Psi \to B_2$*, and* $B_2 <: B_1$*.*

This lemma may not seem intuitive at first glance. Consider a concrete example $\mathsf{Int} \to \forall a.a <: \mathsf{Int} \to \mathsf{Int}$, and $\mathsf{Int} \vdash \mathsf{Int} \to \forall a.a <: \mathsf{Int} \to \forall a.a$. The former one, holds because we have $\forall a.a <: \mathsf{Int}$ in the return type. But in the latter one, after Int is consumed from application context, we eventually reach S-Empty, which always returns the original type back.

3.4 Translation to System F, Coherence and Type-Safety

We translate the source language into a variant of System F that is also used in Peyton Jones et al. [27]. The translation is shown to be coherent and type safe. Due to space limitations, we only summarize the key aspects of the translation. Full details can be found in the supplementary materials of the paper.

The syntax of our target language is as follows:

$$\text{Expressions } s, f ::= x \mid n \mid \lambda x : A.\ s \mid \Lambda a.s \mid s_1\ s_2 \mid s_1\ A$$

In the translation, we use f to refer to the coercion function produced by the subtyping translation, and s to refer to the translated term in System F. We write $\Gamma \vdash^F s : A$ to mean the term s has type A in System F.

The type-directed translation follows the rules in Fig. 1, with a translation output in the forms of judgments. We summarize all judgments as:

Judgment	Translation Output	Soundness
$A <: B \rightsquigarrow f$	coercion function f	$\varnothing \vdash^F f : A \to B$
$\Psi \vdash A <: B \rightsquigarrow f$	coercion function f	$\varnothing \vdash^F f : A \to B$
$\Gamma \mathrel{;} \Psi \vdash e \Rightarrow A \rightsquigarrow s$	target expression s	$\Gamma \vdash^F s : A$

For example, $A <: B \rightsquigarrow f$ means that if $A <: B$ holds in the source language, we can translate it into a System F term f, which is a coercion function and has type $A \to B$. We prove that our system is type safe by proving that the translation produces well-typed terms.

Lemma 9 **(Typing Soundness).** *If* $\Gamma \mathrel{;} \Psi \vdash e \Rightarrow A \rightsquigarrow s$*, then* $\Gamma \vdash^F s : A$*.*

However, there could be multiple targets corresponding to one expression due to the multiple choices for τ. To prove that the translation is coherent, we prove that all the translations for one expression have the same operational semantics. We write $|e|$ for the expressions after type erasure since types are useless after type checking. Because multiple targets could have different number of coercion functions, we use η-id equality [5] instead of syntactic equality, where two expressions are regarded as equivalent if they can turn into the same expression through η-reduction or removal of redundant identity functions. We then prove that our translation actually generates a *unique* target:

Lemma 10 **(Coherence).** *If* $\Gamma_1 \mathrel{;} \Psi_1 \vdash e \Rightarrow A \rightsquigarrow s_1$*, and* $\Gamma_2 \mathrel{;} \Psi_2 \vdash e \Rightarrow B \rightsquigarrow s_2$*, then* $|s_1| \rightsquigarrow_{\eta id} |s_2|$*.*

3.5 Algorithmic System

Even though our specification is syntax-directed, it does not directly lead to an algorithm, because there are still many guesses in the system, such as in rule T-Lam2. This subsection presents a brief introduction of the algorithm, which essentially follows the approach by Peyton Jones et al. [27]. Full details can be found in the supplementary materials.

Instead of guessing, the algorithm creates meta type variables $\widehat{\alpha}, \widehat{\beta}$ which are waiting to be solved. The judgment for the algorithmic type system is $(S_0, N_0) \mid \Gamma \mid \Psi \vdash e \Rightarrow A \hookrightarrow (S_1, N_1)$. Here we use N as name supply, from which we can always extract new names. We use S as a notation for the substitution that maps meta type variables to their solutions. For example, rule T-Lam2 becomes

$$\frac{(S_0, N_0) \mid \Gamma, x : \widehat{\beta} \vdash e \Rightarrow A \hookrightarrow (S_1, N_1)}{(S_0, N_0\widehat{\beta}) \mid \Gamma \vdash \lambda x.\ e \Rightarrow \widehat{\beta} \to A \hookrightarrow (S_1, N_1)} \text{ AT-LAM1}$$

Comparing it to rule T-Lam2, τ is replaced by a new meta type variable $\widehat{\beta}$ from name supply $N_0\widehat{\beta}$. But despite of the name supply and substitution, the rule retains the structure of T-Lam2.

Having the name supply and substitutions, the algorithmic system is a direct extension of the specification in Fig. 1, with a process to do unifications that solve meta type variables. Such unification process is quite standard and similar to the one used in the Hindley-Milner system. We proved our algorithm is sound and complete with respect to the specification.

Theorem 1 (Soundness). *If* $([], N_0) \mid \Gamma \vdash e \Rightarrow A \hookrightarrow (S_1, N_1)$*, then for any substitution* V *with* $dom(V) = fmv\ (S_1\Gamma, S_1A)$*, we have* $VS_1\Gamma \vdash e \Rightarrow VS_1A$*.*

Theorem 2 (Completeness). *If* $\Gamma \vdash e \Rightarrow A$*, then for a fresh* N_0*, we have* $([], N_0) \mid \Gamma \vdash e \Rightarrow B \hookrightarrow (S_1, N_1)$*, and for some* S_2*, we have* $\overline{\Gamma(S_2S_1B)} <: \overline{\Gamma(A)}$*.*

4 More Expressive Type Applications

This section presents a System-F-like calculus, which shows that the application mode not only does work well for calculi with explicit type applications, but it also adds interesting expressive power, while at the same time retaining uniqueness of types for *explicitly* polymorphic functions. One additional novelty in this section is to present another possible variant of typing and subtyping rules for the application mode, by exploiting the lemmas presented in Sects. 3.2 and 3.3.

$$\begin{aligned} \langle \emptyset \rangle A &= A & \langle \Gamma, x : B \rangle A &= \langle \Gamma \rangle A \\ \langle \Gamma, a \rangle A &= \langle \Gamma \rangle A & \langle \Gamma, a = B \rangle A &= \langle \Gamma \rangle (A[\![a \mapsto B]\!]) \end{aligned}$$

Fig. 2. Apply contexts as substitutions on types.

$$\frac{a \in \Gamma}{\Gamma \vdash a}\ \text{WF-TVar} \qquad \frac{}{\Gamma \vdash \mathsf{Int}}\ \text{WF-Int} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}\ \text{WF-Arrow} \qquad \frac{\Gamma, a \vdash A}{\Gamma \vdash \forall a.A}\ \text{WF-All}$$

Fig. 3. Well-formedness.

4.1 Syntax

We focus on a new variant of the standard System F. The syntax is as follows:

$$\begin{array}{ll} \text{Expr} & e ::= x \mid n \mid \lambda x : A.\ e \mid \lambda x.\ e \mid e_1\ e_2 \mid \Lambda a.e \mid e\ [A] \\ \text{Type} & A ::= a \mid \mathsf{Int} \mid A \to B \mid \forall a.A \\ \text{Typing Context} & \Gamma ::= \varnothing \mid \Gamma, x : A \mid \Gamma, a \mid \Gamma, a = A \\ \text{Application Context} & \Psi ::= \varnothing \mid \Psi, A \mid \Psi, [A] \end{array}$$

The syntax is mostly standard. Expressions include variables x, integers n, annotated abstractions $\lambda x : A.\ s$, unannotated abstractions $\lambda x.\ e$, applications $e_1\ e_2$, type abstractions $\Lambda a.s$, and type applications $e_1\ [A]$. Types includes type variable a, integers Int, function types $A \to B$, and polymorphic types $\forall a.A$.

The main novelties are in the typing and application contexts. Typing contexts contain the usual term variable typing $x : A$, type variables a, and type equations $a = A$, which track equalities and are not available in System F. Application contexts use A for the *argument type* for term-level applications, and use $[A]$ for the *type argument itself* for type applications.

Applying Contexts. The typing contexts contain type equations, which can be used as substitutions. For example, $a = \mathsf{Int}, x : \mathsf{Int}, b = \mathit{Bool}$ can be applied to $a \to b$ to get the function type $\mathsf{Int} \to \mathit{Bool}$. We write $\langle \Gamma \rangle A$ for Γ applied as a substitution to type A. The formal definition is given in Fig. 2.

Well-Formedness. The type well-formedness under typing contexts is given in Fig. 3, which is quite straightforward. Notice that there is no rule corresponding to type variables in type equations. For example, a is not a well-formed type under typing context $a = \mathsf{Int}$, instead, $\langle a = \mathsf{Int} \rangle a$ is. In other words, we keep the invariant: *types are always fully substituted under the typing context.*

The well-formedness of typing contexts Γ *ctx*, and the well-formedness of application contexts $\Gamma \vdash \Psi$ can be defined naturally based on the well-formedness of types. The specific definitions can be found in the supplementary materials.

$$\boxed{\Gamma \shortmid \Psi \vdash e \;\Rightarrow\; B}$$

$$\frac{\Gamma\ ctx \qquad \Gamma \vdash \Psi \qquad x : A \in \Gamma \qquad \Psi \vdash A \;<:\; B}{\Gamma \shortmid \Psi \vdash x \;\Rightarrow\; B}\ \textsc{SF-Var} \qquad \frac{\Gamma\ ctx}{\Gamma \vdash n \;\Rightarrow\; \mathsf{Int}}\ \textsc{SF-Int}$$

$$\frac{\Gamma, x : \langle\Gamma\rangle A \vdash e \;\Rightarrow\; B}{\Gamma \vdash \lambda x : A.\ e \;\Rightarrow\; \langle\Gamma\rangle A \rightarrow B}\ \textsc{SF-LamAnn1}$$

$$\frac{\Gamma, x : \langle\Gamma\rangle A \shortmid \Psi \vdash e \;\Rightarrow\; B}{\Gamma \shortmid \Psi, \langle\Gamma\rangle A \vdash \lambda x : A.\ e \;\Rightarrow\; B}\ \textsc{SF-LamAnn2} \qquad \frac{\Gamma, x : A \shortmid \Psi \vdash e \;\Rightarrow\; B}{\Gamma \shortmid \Psi, A \vdash \lambda x.\ e \;\Rightarrow\; B}\ \textsc{SF-Lam}$$

$$\frac{\Gamma \vdash e_2 \;\Rightarrow\; A \qquad \Gamma \shortmid \Psi, A \vdash e_1 \;\Rightarrow\; B}{\Gamma \shortmid \Psi \vdash e_1\ e_2 \;\Rightarrow\; B}\ \textsc{SF-App} \qquad \frac{\Gamma, a \vdash e \;\Rightarrow\; B}{\Gamma \vdash \Lambda a.e \;\Rightarrow\; \forall a.B}\ \textsc{SF-TLam1}$$

$$\frac{\Gamma, a = A \shortmid \Psi \vdash e \;\Rightarrow\; B}{\Gamma \shortmid \Psi, [A] \vdash \Lambda a.e \;\Rightarrow\; B}\ \textsc{SF-TLam2} \qquad \frac{\Gamma \shortmid \Psi, [\langle\Gamma\rangle A] \vdash e \;\Rightarrow\; B}{\Gamma \shortmid \Psi \vdash e\ [A] \;\Rightarrow\; B}\ \textsc{SF-TApp}$$

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

$$\frac{}{\varnothing \vdash A \;<:\; A}\ \textsc{SF-SEmpty}$$

$$\frac{\Psi \vdash B[\![a \mapsto A]\!] \;<:\; C}{\Psi, [A] \vdash \forall a.B \;<:\; C}\ \textsc{SF-STApp} \qquad \frac{\Psi \vdash B \;<:\; C}{\Psi, A \vdash A \rightarrow B \;<:\; C}\ \textsc{SF-SApp}$$

Fig. 4. Type system for the new System F variant.

4.2 Type System

Typing Judgments. From Lemmas 1 and 4, we know that the application context always coincides with typing/subtyping results. This means that the types of the arguments can be recovered from the application context. So instead of the whole type, we can use only the return type as the output type. For example, we review the rule T-Lam in Fig. 1:

$$\frac{\Gamma, x : A \shortmid \Psi \vdash e \;\Rightarrow\; B}{\Gamma \shortmid \Psi, A \vdash \lambda x.\ e \;\Rightarrow\; A \rightarrow B}\ \textsc{T-Lam} \qquad \frac{\Gamma, x : A \shortmid \Psi \vdash e \;\Rightarrow\; C}{\Gamma \shortmid \Psi, A \vdash \lambda x.\ e \;\Rightarrow\; C}\ \textsc{T-Lam-Alt}$$

We have $B = \Psi \rightarrow C$ for some C by Lemma 1. Instead of B, we can directly return C as the output type, since we can derive from the application context that e is of type $\Psi \rightarrow C$, and $\lambda x.\ e$ is of type $(\Psi, A) \rightarrow C$. Thus we obtain the T-Lam-Alt rule.

Note that the choice of the style of the rules is only a matter of taste in the language in Sect. 3. However, it turns out to be very useful for our variant of System F, since it helps avoiding introducing types like $\forall a = \mathsf{Int}.a$. Therefore, we adopt the new form of judgment. Now the judgment $\Gamma \shortmid \Psi \vdash e \;\Rightarrow\; A$ is interpreted as: *under the typing context* Γ, *and the application context* Ψ, *the return type of* e *applied to the arguments whose types are in* Ψ *is* A.

Typing Rules. Using the new interpretation of the typing judgment, we give the typing rules in the top of Fig. 4. SF-Var depends on the subtyping rules. Rule SF-Int always infers integer types. Rule SF-LamAnn1 first applies current context on A, then puts $x : \langle \Gamma \rangle A$ into the typing context to infer e. The return type is a function type because the application context is empty. Rule SF-LamAnn2 has a non-empty application context, so it requests that the type at the top of the application context is equivalent to $\langle \Gamma \rangle A$. The output type is B instead of a function type. Notice how the invariant that types are fully substituted under the typing context is preserved in these two rules.

Rule SF-Lam pops the type A from the application context, puts $x : A$ into the typing context, and returns only the return type B. In rule SF-App, the argument type A is pushed into the application context for inferring e_1, so the output type B is the type of e_1 under application context (Ψ, A), which is exactly the return type of $e_1\ e_2$ under Ψ.

Rule SF-TLam1 is for type abstractions. The type variable a is pushed into the typing context, and the return type is a polymorphic type. In rule SF-TLam2, the application context has the type argument A at its top, which means the type abstraction is applied to A. We then put the type equation $a = A$ into the typing context to infer e. Like term-level applications, here we only return the type B instead of a polymorphic type. In rule SF-TApp, we first apply the typing context on the type argument A, then we put the applied type argument $\langle \Gamma \rangle A$ into the application context to infer e, and return B as the output type.

Subtyping. The definition of subtyping is given at the bottom of Fig. 4. As with the typing rules, the part of argument types corresponding to the application context is omitted in the output. We interpret the rule form $\Psi \vdash A <: B$ as, under the application context Ψ, A is a subtype of the type whose type arguments are Ψ and the return type is B.

Rule SF-SEmpty returns the input type under the empty application context. Rule SF-STApp instantiates a with the type argument A, and returns C. Note how application subtyping can be extended naturally to deal with type applications. Rule SF-SApp requests that the argument type is the same as the top type in the application context, and returns C.

4.3 Meta Theory

Applying the idea of the application mode to System F results in a well-behaved type system. For example, subtyping transitivity becomes more concise:

Lemma 11 (Subtyping transitivity). *If $\Psi_1 \vdash A <: B$, and $\Psi_2 \vdash B <: C$, then $\Psi_2, \Psi_1 \vdash A <: C$.*

Also, we still have the interesting subsumption lemma that transfers from the inference mode to the application mode:

Lemma 12 (Subsumption). *If $\Gamma \vdash e \Rightarrow A$, and $\Gamma \vdash \Psi$, and $\Psi \vdash A <: B$, then $\Gamma \mid \Psi \vdash e \Rightarrow B$.*

Furthermore, we prove the type safety by proving the progress lemma and the preservation lemma. The detailed definitions of operational semantics and values can be found in the supplementary materials.

Lemma 13 (Progress). *If $\varnothing \vdash e \Rightarrow T$, then either e is a value, or there exists e', such that $e \longrightarrow e'$.*

Lemma 14 (Preservation). *If $\Gamma \shortmid \Psi \vdash e \Rightarrow A$, and $e \longrightarrow e'$, then $\Gamma \shortmid \Psi \vdash e' \Rightarrow A$.*

Moreover, introducing type equality preserves unique types:

Lemma 15 (Uniqueness of typing). *If $\Gamma \shortmid \Psi \vdash e \Rightarrow A$, and $\Gamma \shortmid \Psi \vdash e \Rightarrow B$, then $A = B$.*

5 Discussion

This section discusses possible design choices regarding bi-directional type checking with the application mode, and talks about possible future work.

5.1 Combining Application and Checked Modes

Although the application mode provides us with alternative design choices in a bi-directional type system, a checked mode can still be *easily* added. One motivation for the checked mode would be annotated expressions $e : A$, where the type of expressions is known and is therefore used to check expressions.

Consider adding $e : A$ for introducing the third checked mode for the language in Sect. 3. Notice that, since the checked mode is stronger than application mode, when entering checked mode the application context is no longer useful. Instead we use application subtyping to satisfy the application context requirements. A possible typing rule for annotation expressions is:

$$\frac{\Psi \vdash A <: B \qquad \Gamma \vdash e \Leftarrow A}{\Gamma \shortmid \Psi \vdash (e : A) \Rightarrow B}\ \textsc{T-Ann}$$

Here, e is checked using its annotation A, and then we instantiate A to B using subtyping with application context Ψ.

Now we can have a rule set of the checked mode for all expressions. For example, one useful rule for abstractions in checked mode could be Abs-Chk, where the parameter type A serves as the type of x, and typing checks the body with B. Also, combined with the information flow, the checked rule for application checks the function with the full type.

$$\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\ e \Leftarrow A \to B}\ \textsc{Abs-Chk} \qquad \frac{\Gamma \vdash e_2 \Rightarrow A \qquad \Gamma \vdash e_1 \Leftarrow A \to B}{\Gamma \vdash e_1\ e_2 \Leftarrow B}\ \textsc{App-Chk}$$

Note that adding expression annotations might bring convenience for programmers, since annotations can be more freely placed in a program. For example, `(λf. f 1) : (Int → Int) →Int` becomes valid. However this does not add expressive power, since programs that are typeable under expression annotations, would remain typeable after moving the annotations to bindings. For example the previous program is equivalent to `(λf : (Int → Int). f 1)`.

This discussion is a sketch. We have not defined the corresponding declarative system nor algorithm. However we believe that the addition of a checked mode will *not* bring surprises to the meta-theory.

5.2 Additional Constructs

In this section, we show that the application mode is compatible with other constructs, by discussing how to add support for pairs in the language given in Sect. 3. A similar methodology would apply to other constructs like sum types, data types, if-then-else expressions and so on.

The introduction rule for pairs must be in the inference mode with an empty application context. Also, the subtyping rule for pairs is as expected.

$$\frac{\Gamma \vdash e_1 \;\Rightarrow\; A \qquad \Gamma \vdash e_2 \;\Rightarrow\; B}{\Gamma \vdash (e_1, e_2) \;\Rightarrow\; (A, B)}\ \textsc{T-Pair} \qquad \frac{A_1 \;<:\; B_1 \qquad A_2 \;<:\; B_2}{(A_1, A_2) \;<:\; (B_1, B_2)}\ \textsc{S-Pair}$$

The application mode can apply to the elimination constructs of pairs. If one component of the pair is a function, for example, $(\textbf{fst}\ (\lambda x.\ x, 3)\ 4)$, then it is possible to have a judgment with a non-empty application context. Therefore, we can use the application subtyping to account for the application contexts:

$$\frac{\Gamma \vdash e \;\Rightarrow\; (A, B) \qquad \Psi \vdash A \;<:\; C}{\Gamma \,\shortmid\, \Psi \vdash \textbf{fst}\ e \;\Rightarrow\; C}\ \textsc{T-Fst1} \qquad \frac{\Gamma \vdash e \;\Rightarrow\; (A, B) \qquad \Psi \vdash B \;<:\; C}{\Gamma \,\shortmid\, \Psi \vdash \textbf{snd}\ e \;\Rightarrow\; C}\ \textsc{T-Snd1}$$

However, in polymorphic type systems, we need to take the subsumption rule into consideration. For example, in the expression $(\lambda x : (\forall a.(a, b)).\ \textbf{fst}\ x)$, **fst** is applied to a polymorphic type. Interestingly, instead of a non-deterministic subsumption rule, having polymorphic types actually leads to a simpler solution. According to the philosophy of the application mode, the types of the arguments always flow into the functions. Therefore, instead of regarding $(\textbf{fst}\ e)$ as an expression form, where e is itself an argument, we could regard **fst** as a function on its own, whose type is $(\forall ab.(a, b) \to a)$. Then as in the variable case, we use the subtyping rule to deal with application contexts. Thus the typing rules for **fst** and **snd** can be modeled as:

$$\frac{\Psi \vdash (\forall ab.(a, b) \to a) \;<:\; A}{\Gamma \,\shortmid\, \Psi \vdash \textbf{fst} \;\Rightarrow\; A}\ \textsc{T-Fst2} \qquad \frac{\Psi \vdash (\forall ab.(a, b) \to b) \;<:\; A}{\Gamma \,\shortmid\, \Psi \vdash \textbf{snd} \;\Rightarrow\; A}\ \textsc{T-Snd2}$$

Note that another way to model those two rules would be to simply have an initial typing environment $\Gamma_{initial} \equiv \textbf{fst} : (\forall ab.(a, b) \to a), \textbf{snd} : (\forall ab.(a, b) \to b)$. In this case the elimination of pairs be dealt directly by the rule for variables.

An extended version of the calculus presented in Sect. 3, which includes the rules for pairs (T-Pair, S-Pair, T-Fst2 and T-Snd2), has been formally studied. All the theorems presented in Sect. 3 hold with the extension of pairs.

5.3 Dependent Type Systems

One remark about the application mode is that the same idea is possibly applicable to systems with advanced features, where type inference is sophisticated or even undecidable. One promising application is, for instance, dependent type systems [2,3,10,21,37]. Type systems with dependent types usually unify the syntax for terms and types, with a single lambda abstraction generalizing both type and lambda abstractions. Unfortunately, this means that the **let** desugar is not valid in those systems. As a concrete example, consider desugaring the expression $\mathbf{let}\, a = \mathsf{Int}\, \mathbf{in}\, \lambda x : a.\ x + 1$ into `(λa. λx : a. x + 1) Int`, which is ill-typed because the type of x in the abstraction body is a and not Int.

Because **let** cannot be encoded, declarations cannot be encoded either. Modeling declarations in dependently typed languages is a subtle matter, and normally requires some additional complexity [34].

We believe that the same technique presented in Sect. 4 can be adapted into a dependently typed language to enable a **let** encoding. In a dependent type system with unified syntax for terms and types, we can combine the two forms in the typing context ($x : A$ and $a = A$) into a unified form $x = e : A$. Then we can combine two application rules SF-App and SF-TApp into De-App, and also two abstraction rules SF-Lam and SF-TLam1 into De-Lam.

$$\frac{\Gamma \vdash e_2 \ \Rightarrow\ A \qquad \Gamma \mid \Psi, e_2 : A \vdash e_1 \ \Rightarrow\ B}{\Gamma \mid \Psi \vdash e_1\ e_2 \ \Rightarrow\ B}\ \textsc{De-App} \qquad \frac{\Gamma, x = e_1 : A \mid \Psi \vdash e \ \Rightarrow\ B}{\Gamma \mid \Psi, e_1 : A \vdash \lambda x.\ e \ \Rightarrow\ B}\ \textsc{De-Lam}$$

With such rules it would be possible to handle declarations easily in dependent type systems. Note this is still a rough idea and we have not fully worked out the typing rules for this type system yet.

6 Related Work

6.1 Bi-directional Type Checking

Bi-directional type checking was popularized by the work of Pierce and Turner [29]. It has since been applied to many type systems with advanced features. The alternative application mode introduced by us enables a variant of bi-directional type checking. There are many other efforts to refine bi-directional type checking.

Colored local type inference [25] refines local type inference for *explicit* polymorphism by propagating partial type information. Their work is built on distinguishing inherited types (known from the context) and synthesized types (inferred from terms). A similar distinction is achieved in our algorithm by manipulating type variables [14]. Also, their information flow is from functions to arguments, which is fundamentally different from the application mode.

The system of *tridirectional* type checking [15] is based on bi-directional type checking and has a rich set of property types including intersections, unions and quantified dependent types, but without parametric polymorphism. Tridirectional type checking has a new direction for supporting type checking unions and existential quantification. Their third mode is basically unrelated to our application mode, which propagates information from outer applications.

Greedy bi-directional polymorphism [13] adopts a greedy idea from Cardelli [4] on bi-directional type checking with higher ranked types, where the type variables in instantiations are determined by the first constraint. In this way, they support some uses of impredicative polymorphism. However, the greediness also makes many obvious programs rejected.

6.2 Type Inference for Higher-Ranked Types

As a reference, Fig. 5 [14,20] gives a high-level comparison between related works and our system.

Predicative Systems. Peyton Jones et al. [27] developed an approach for type inference for higher rank types using traditional bi-directional type checking based on Odersky and Läufer [24]. However in their system, in order to do instantiation on higher rank types, they are forced to have an additional type category (ρ types) as a special kind of higher rank type without top-level quantifiers. This complicates their system since they need to have additional rule sets for such types. They also combine a variant of the containment relation from Mitchell [23] for deep skolemisation in subsumption rules, which we believe is compatible with our subtyping definition.

Dunfield and Krishnaswami [14] build a simple and concise algorithm for higher ranked polymorphism based on traditional bidirectional type checking. They deal with the same language of Peyton Jones et al. [27], except they do not have *let* expressions nor generalization (though it is discussed in design variations). They have a special *application judgment* which delays instantiation until the expression is applied to some argument. As with application mode, this avoids the additional category of types. Unlike their work, our work supports generalization and HM-style *let* expressions. Moreover the use of an application mode in our work introduces several differences as to when and where annotations are needed (see Sect. 2.4 for related discussion).

Impredicative Systems. ML^F [18,19,32] generalizes ML with first-class polymorphism. ML^F introduces a new type of bounded quantification (either rigid or flexible) for polymorphic types so that instantiation of polymorphic bindings is delayed until a principal type is found. The HML system [20] is proposed as a simplification and restriction of ML^F. HML only uses flexible types, which simplifies the type inference algorithm, but retains many interesting properties and features.

The FPH system [35] introduces boxy monotypes into System F types. One critique of boxy type inference is that the impredicativity is deeply hidden in the algorithmic type inference rules, which makes it hard to understand the interaction between its predicative constraints and impredicative instantiations [31].

System	Types	Impred	Let	Annotations
ML^F	flexible and rigid	yes	yes	on polymorphically used parameters
HML	flexible F-types	yes	yes	on polymorphic parameters
FPH	boxy F-types	yes	yes	on polymorphic parameters and some let bindings with higher-ranked types
Peyton Jones et al. (2007)	F-types	no	yes	on polymorphic parameters
Dunfield et al. (2013)	F-types	no	no	on polymorphic parameters
this paper	F-types	no	sugar	on polymorphic parameters that are not applied

Fig. 5. Comparison of higher-ranked type inference systems.

6.3 Tracking Type Equalities

Tracking type equalities is useful in various situations. Here we discuss specifically two related cases where tracking equalities plays an important role.

Type Equalities in Type Checking. Tracking type equalities is one essential part for type checking algorithms involving Generalized Algebraic Data Types (GADTs) [6,26,33]. For example, Peyton Jones et al. [26] propose a type inference algorithm based on unification for GADTs, where type equalities only apply to user-specified types. However, reasoning about type equalities in GADTs is essentially different from the approach in Sect. 4: type equalities are introduced by pattern matches in GADTs, while they are introduced through type applications in our system. Also, type equalities in GADTs are local, in the sense different branches in pattern matches have different type equalities for the same type variable. In our system, a type equality is introduced globally and is never changed. However, we believe that they can be made compatible by distinguishing different kinds of equalities.

Equalities in Declarations. In systems supporting dependent types, type equalities can be introduced by declarations. In the variant of pure type systems proposed by Severi and Poll [34], expressions $x = a : A \textbf{ in } b$ generate an equality $x = a : A$ in the typing context, which can be fetched later through δ-reduction. However, δ-reduction rules require careful design, and the conversion rule of δ-reduction makes the type system non-deterministic. One potential usage of the application mode is to help reduce the complexity for introducing declarations in those type systems, as briefly discussed in Sect. 5.3.

7 Conclusion

We proposed a variant of bi-directional type checking with a new *application mode*, where type information flows from arguments to functions in applications. The application mode is essentially a generalization of the inference mode, can therefore work naturally with inference mode, and avoid the rule duplication that is often needed in traditional bi-directional type checking. The application mode can also be combined with the checked mode, but this often does not add expressiveness. Compared to traditional bi-directional type checking, the application mode opens a new path to the design of type inference/checking.

We have adopted the application mode in two type systems. Those two systems enjoy many interesting properties and features. However as bi-directional type checking can be applied to many type systems, we believe application mode is applicable to various type systems. One obvious potential future work is to investigate more systems where the application mode brings benefits. This includes systems with subtyping, intersection types [8,30], static overloading, or dependent types.

Acknowledgements. We thank 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. Abel, A.: Termination checking with types. RAIRO-Theor. Inform. Appl. **38**(4), 277–319 (2004)
2. Abel, A., Coquand, T., Dybjer, P.: Verifying a semantic $\beta\eta$-conversion test for Martin-Löf type theory. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 29–56. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70594-9_4
3. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: A bi-directional refinement algorithm for the calculus of (co) inductive constructions. Log. Meth. Comput. Sci. **8**, 1–49 (2012)
4. Cardelli, L.: An implementation of FSub. Technical report, Research report 97. Digital Equipment Corporation Systems Research Center (1993)
5. Chen, G.: Coercive subtyping for the calculus of constructions. In: POPL 2003 (2003)
6. Cheney, J., Hinze, R.: First-class phantom types. Technical Report CUCIS TR2003-1901. Cornell University (2003)
7. Chlipala, A., Petersen, L., Harper, R.: Strict bidirectional type checking. In: International Workshop on Types in Languages Design and Implementation (2005)
8. Coppo, M., Dezani-Ciancaglini, M., Venneri, B.: Functional characters of solvable terms. Math. Log. Q. **27**(2–6), 45–58 (1981)
9. Coq Development Team: The Coq proof assistant, Documentation, system download (2015)
10. Coquand, T.: An algorithm for type-checking dependent types. Sci. Comput. Program. **26**(1–3), 167–177 (1996)
11. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL 1982 (1982)
12. Davies, R., Pfenning, F.: Intersection types and computational effects. In: ICFP 2000 (2000)
13. Dunfield, J.: Greedy bidirectional polymorphism. In: Workshop on ML (2009)
14. Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. In: ICFP 2013 (2013)

15. Dunfield, J., Pfenning, F.: Tridirectional typechecking. In: POPL 2004 (2004)
16. Girard, J.-Y.: The system F of variable types, fifteen years later. Theor. Comput. Sci. **45**, 159–192 (1986)
17. Hindley, J.R.: The principal type-scheme of an object in combinatory logic. Trans. Am. Math. Soc. **146**, 29–60 (1969)
18. Le Botlan, D., Rémy, D.: MLF: Raising ML to the power of system F. In: ICFP 2003 (2003)
19. Le Botlan, D., Rémy, D.: Recasting MLF. Inform. Comput. **207**(6), 726–785 (2009)
20. Leijen, D.: Flexible types: robust type inference for first-class polymorphism. In: POPL 2009 (2009)
21. Löh, A., McBride, C., Swierstra, W.: A tutorial implementation of a dependently typed lambda calculus. Fundamenta informaticae **102**(2), 177–207 (2010)
22. Lovas, W.: Refinement types for logical frameworks. Ph.D. thesis, Carnegie Mellon University (2010). AAI3456011
23. Mitchell, J.C.: Polymorphic type inference and containment. Inform. Comput. **76**(2–3), 211–249 (1988)
24. Odersky, M., Läufer, K.: Putting type annotations to work. In: POPL 1996 (1996)
25. Odersky, M., Zenger, C., Zenger, M.: Colored local type inference. In: POPL 2001 (2001)
26. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for gadts. In: ICFP 2006 (2006)
27. Peyton Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. J. Funct. Program. **17**(01), 1–82 (2007)
28. Pientka, B.: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: POPL 2008 (2008)
29. Pierce, B.C., Turner, D.N.: Local type inference. TOPLAS **22**(1), 1–44 (2000)
30. Pottinger, G.: A type assignment for the strongly normalizable λ-terms. In: To HB Curry: essays on combinatory logic, lambda calculus and formalism. pp. 561–577 (1980)
31. Rémy, D.: Simple, partial type-inference for system F based on type-containment. In: ICFP 2005 (2005)
32. Rémy, D., Yakobowski, B.: From ML to MLF: graphic type constraints with efficient type inference. In: ICFP 2008 (2008)
33. Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for gadts. In: ICFP 2009 (2009)
34. Severi, P., Poll, E.: Pure type systems with definitions. In: Nerode, A., Matiyasevich, Y.V. (eds.) LFCS 1994. LNCS, vol. 813, pp. 316–328. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58140-5_30
35. Vytiniotis, D., Weirich, S., Peyton Jones, S.: FPH: First-class polymorphism for haskell. In: ICFP 2008 (2008)
36. Wells, J.B.: Typability and type checking in system F are equivalent and undecidable. Ann. Pure Appl. Log. **98**(1–3), 111–156 (1999)
37. Xi, H., Pfenning, F.: Dependent types in practical programming. In: POPL 1999 (1999)

Behavioural Equivalence via Modalities for Algebraic Effects

Alex Simpson and Niels Voorneveld(✉)

Faculty of Mathematics and Physics, University of Ljubljana, Ljubljana, Slovenia
{Alex.Simpson,Niels.Voorneveld}@fmf.uni-lj.si

Abstract. The paper investigates behavioural equivalence between programs in a call-by-value functional language extended with a signature of (algebraic) effect-triggering operations. Two programs are considered as being behaviourally equivalent if they enjoy the same behavioural properties. To formulate this, we define a logic whose formulas specify behavioural properties. A crucial ingredient is a collection of *modalities* expressing effect-specific aspects of behaviour. We give a general theory of such modalities. If two conditions, *openness* and *decomposability*, are satisfied by the modalities then the logically specified behavioural equivalence coincides with a modality-defined notion of applicative bisimilarity, which can be proven to be a congruence by a generalisation of Howe's method. We show that the openness and decomposability conditions hold for several examples of algebraic effects: nondeterminism, probabilistic choice, global store and input/output.

1 Introduction

The notion of *behavioural equivalence* between programs is a fundamental concept in the theory of programming languages. A conceptually natural approach to defining behavioural equivalence is to consider two programs as being equivalent if they enjoy the same 'behavioural properties'. This can be made precise by specifying a *behavioural logic* whose formulas express behavioural properties. Two programs M, N are then defined to be equivalent if, for all formulas Φ, it holds that $M \models \Phi$ iff $N \models \Phi$ (where $M \models \Phi$ expresses the satisfaction relation: program M enjoys property Φ).

This logical approach to defining behavioural equivalence has been particularly prominent in concurrency theory, where the classic result is that the equivalence defined by Hennessy-Milner logic [4] coincides with bisimilarity [14,17]. The aim of the present paper is to adapt the logical approach to the very different computational paradigm of *applicative programming with effects*.

A. Simpson—Supported by the Slovenian Research Agency, research core funding No. P1–0294.

N. Voorneveld—Supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326, and by EU-MSCA-RISE project 731143 (CID).

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 300–326, 2018.
https://doi.org/10.1007/978-3-319-89884-1_11

More precisely, we consider a call-by-value functional programming language with *algebraic effects* in the sense of Plotkin and Power [21]. Broadly speaking, effects are those aspects of computation that involve a program interacting with its 'environment'; for example: nondeterminism, probabilistic choice (in both cases, the choice is deferred to the environment); input/output; mutable store (the machine state is modified); control operations such as exceptions, jumps and handlers (which interact with the continuation in the evaluation process); etc. Such general effects collectively enjoy common properties identified in the work of Moggi on monads [15]. Among them, algebraic effects play a special role. They can be included in a programming language by adding effect-triggering operations, whose 'algebraic' nature means that effects act independently of the continuation. From the aforementioned examples of effects, only jumps and handlers are non-algebraic. Thus the notion of algebraic effect covers a broad range of effectful computational behaviour. Call-by-value functional languages provide a natural context for exploring effectful programming. From a theoretical viewpoint, other programming paradigms are subsumed; for example, imperative programs can be recast as effectful functional ones. From a practical viewpoint, the combination of effects with call-by-value leads to the natural programming style supported by impure functional languages such as OCaml.

In order to focus on the main contributions of the paper (the behavioural logic and its induced behavioural equivalence), we instantiate "call-by-value functional language with algebraic effects" using a very simple language. Our language is a simply-typed λ-calculus with a base type of natural numbers, general recursion, call-by-value function evaluation, and algebraic effects, similar to [21]; although, for technical convenience, we adopt the (equivalent) formulation of fine-grained call-by-value [13]. The language is defined precisely in Sect. 2. Following [8,21], an operational semantics is given that evaluates programs to *effect trees*.

Section 3 introduces the behavioural logic. In our impure functional setting, the evaluation of a program of type τ results in a computational process that may or may not invoke effects, and which may or may not terminate with a return *value* of type τ. The key ingredient in our logic is an effect-specific family $\mathcal{O}$ of *modalities*, where each modality $o \in \mathcal{O}$ converts a property ϕ of values of type τ to a property $o\,\phi$ of general programs (called *computations*) of type τ. The idea is that such modalities capture all relevant effect-specific behavioural properties of the effects under consideration.

A main contribution of the paper is to give a general framework for defining such effect modalities, applicable across a wide range of algebraic effects. The general setting is that we have a signature Σ of effect operations, which determines the programming language, and a collection $\mathcal{O}$ of modalities, which determines the behavioural logic. In order to specify the semantics of the logic, we require each modality to be assigned a set of unit-type effect trees, which determines the meaning of the modality. Several concrete examples and a detailed general explanation are given in Sect. 3.

In Sect. 4, we consider the relation of *behavioural equivalence* between programs determined by the logic. A fundamental well-behavedness property is that any reasonable program equivalence should be a congruence with respect to the syntactic constructs of the programming language. Our main theorem (Theorem 1) is that, under two conditions on the collection $\mathcal{O}$ of modalities, which hold for all the examples of effects we consider, the logically induced behavioural equivalence is indeed a congruence.

In order to prove Theorem 1, we develop an alternative perspective on behavioural equivalence, which is of interest in its own right. In Sect. 5 we show how the modalities $\mathcal{O}$ determine a relation of *applicative* $\mathcal{O}$*-bisimilarity*, which is an effect-sensitive version of Abramsky's notion of *applicative bisimilarity* [1]. Theorem 2 shows that applicative $\mathcal{O}$-bisimilarity coincides with the logically defined relation of behavioural equivalence.

The proof of Theorem 1 is then concluded in Sect. 6, where we use Howe's method [5,6] to show that applicative $\mathcal{O}$-bisimilarity is a congruence. Although the proof is technically involved, we give only a brief outline, as the details closely follow the recent paper [9], in which Howe's method is applied to an untyped language with general algebraic effects.

In Sect. 7, we present a variation on our behavioural logic, in which we make the syntax of logical formulas independent of the syntax of the programming language.

Finally, in Sect. 8 we discuss related and further work.

2 A Simple Programming Language

As motivated in the introduction, our chosen base language is a simply-typed call-by-value functional language with general recursion and a ground type of natural numbers, to which we add (algebraic) effect-triggering operations. This means that our language is a call-by-value variant of PCF [20], extended with algebraic effects, resulting in a language similar to the one considered in [21]. In order to simplify the technical treatment of the language, we present it in the style of *fine-grained call-by-value* [13]. This means that we make a syntactic distinction between *values* and *computations*, representing the static and dynamic aspects of the language respectively. Furthermore, all *sequencing* of computations is performed using a single language construct, the **let** construct. The resulting language is straightforwardly intertranslatable with the more traditional call-by-value formulation. But the encapsulation of all sequencing within a single construct has the benefit of avoiding redundancy in proofs.

Our types are just the simple types obtained by iterating the function type construction over two base types: $\mathbf{N}$ of natural numbers, and also a unit type $\mathbf{1}$.

Types: $\tau, \rho ::= \ \mathbf{1} \mid \mathbf{N} \mid \rho \to \tau$
Contexts: $\Gamma ::= \ \emptyset \mid \Gamma, x : \tau$

As usual, term variables x are taken from a countably-infinite stock of such variables, and the context $\Gamma, x : \tau$ can only be formed if the variable x does not already appear in Γ.

As discussed above, program terms are separated into two mutually defined but disjoint categories: *values* and *computations*.

Values: $V, W ::= \ * \mid Z \mid S(V) \mid \lambda x.M \mid x$
Computations: $M, N ::= \ VW \mid$ **return** $V \mid$ **let** $\ M \Rightarrow x$ **in** $\ N \mid$ **fix** $(V) \mid$
case V **in** $\{Z \Rightarrow M, S(x) \Rightarrow N\}$

Here, $*$ is the unique value of the unit type. The values of the type of natural numbers are the *numerals* represented using zero Z and successor S. The values of function type are the λ-abstractions. And a variable x can be considered a value, because, under the call-by-value evaluation strategy of the language, it can only be instantiated with a value.

The computations are: function application VW; the computation that does nothing but return a value V; a **let** construct for sequencing; a **fix** construct for recursive definition; and a **case** construct that branches according to whether its natural-number argument is zero or positive. The computation **let** $M \Rightarrow x$ **in** N implements sequencing in the following sense. First the computation M is evaluated. Only in the case that the evaluation of M terminates, with return value V, does the thread of execution continue to N. In this case, the computation $N[V/x]$ is evaluated, and its return value (if any) is the one returned by the **let** construct.

To the pure functional language described above, we add *effect operations.* The collection of effect operations is specified by a set Σ (the *signature*) of such operations, together with, for each $\sigma \in \Sigma$ an associated *arity* which takes one of the four forms below

$$\alpha^n \to \alpha \qquad \mathbf{N} \times \alpha^n \to \alpha \qquad \alpha^{\mathbf{N}} \to \alpha \qquad \mathbf{N} \times \alpha^{\mathbf{N}} \to \alpha.$$

The notation here is chosen to be suggestive of the way in which such arities are used in the typing rules below, viewing α as a type variable. Each of the forms of arity has an associated term constructor, for building additional computation terms, with which we extend the above grammar for computation terms.

Effects: $\sigma(M_0, M_1, \ldots, M_{n-1}) \mid \sigma(V; M_0, M_1, \ldots, M_{n-1}) \mid \sigma(V) \mid \sigma(W; V)$

Motivating examples of effect operations and their computation terms can be found in Examples 0–5 below.

The typing rules for the language are given in Fig. 1 below. Note that the choice of typing rule for an effect operation $\sigma \in \Sigma$ depends on its declared arity.

The terms of type τ are the values and computations generated by the constructors above. Every term has a unique *aspect* as either a value or computation. We write $Val(\tau)$ and $Com(\tau)$ respectively for closed values and computations. So the closed terms of τ are $Term(\tau) = Val(\tau) \cup Com(\tau)$. For $n \in \mathbb{N}$ a natural number, we write $\overline{n}$ for the numeral $S^n(Z)$, hence $Val(\mathbf{N}) := \{\overline{n} \mid n \in \mathbb{N}\}$.

We now consider some standard signatures of computationally interesting effect operations, which will be used as running examples throughout the paper. (We use the same examples as in [8].)

Example 0 (Pure functional computation). This is the trivial case (from an effect point of view) in which the signature Σ of effect operations is empty. The resulting language is a call-by-value variant of PCF [20].

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{}{\Gamma \vdash * : \mathbf{1}} \qquad \frac{}{\Gamma \vdash Z : \mathbf{N}} \qquad \frac{\Gamma \vdash V : \mathbf{N}}{\Gamma \vdash S(V) : \mathbf{N}}$$

$$\frac{\Gamma \vdash V : \tau}{\Gamma \vdash \mathbf{return}(V) : \tau} \qquad \frac{\Gamma, x : \tau \vdash M : \rho}{\Gamma \vdash (\lambda x : \tau . M) : \tau \to \rho}$$

$$\frac{\Gamma \vdash V : \tau \to \rho \qquad \Gamma \vdash W : \tau}{\Gamma \vdash (VW) : \rho} \qquad \frac{\Gamma \vdash V : (\tau \to \rho) \to (\tau \to \rho)}{\Gamma \vdash \mathbf{fix}(V) : \tau \to \rho}$$

$$\frac{\Gamma \vdash V : \mathbf{N} \qquad \Gamma \vdash M : \tau \qquad \Gamma, x : \mathbf{N} \vdash N : \tau}{\Gamma \vdash \ \mathbf{case}\ V\ \mathbf{of}\ \{Z \Rightarrow M; S(x) \Rightarrow N\} : \tau} \quad \frac{\Gamma \vdash M : \tau \qquad \Gamma, x : \tau \vdash N : \rho}{\Gamma \vdash \mathbf{let}\ M \Rightarrow x\ \mathbf{in}\ N : \rho}$$

$$\frac{\sigma : \alpha^n \to \alpha \qquad \Gamma \vdash M_i : \tau}{\Gamma \vdash \sigma(M_0, M_1, \ldots, M_{n-1}) : \tau} \qquad \frac{\sigma : \alpha^{\mathbf{N}} \to \alpha \qquad \Gamma \vdash V : \mathbf{N} \to \tau}{\Gamma \vdash \sigma(V) : \tau}$$

$$\frac{\sigma : \mathbf{N} \times \alpha^n \to \alpha \qquad \Gamma \vdash V : \mathbf{N} \qquad \Gamma \vdash M_i : \tau}{\Gamma \vdash \sigma(V; M_0, M_1, \ldots, M_{n-1}) : \tau}$$

$$\frac{\sigma : \mathbf{N} \times \alpha^{\mathbf{N}} \to \alpha \qquad \Gamma \vdash V : \mathbf{N} \qquad \Gamma \vdash W : \mathbf{N} \to \tau}{\Gamma \vdash \sigma(V; W) : \tau}$$

Fig. 1. Typing rules

Example 1 (Error). We take a set of error labels E. For each $e \in E$ there is an effect operator $\mathit{raise}_e : \alpha^0 \to \alpha$ which, when invoked by the computation $\mathit{raise}_e()$, aborts evaluation and outputs e as an error message.

Example 2 (Nondeterminism). There is a binary choice operator $\mathit{or} : \alpha^2 \to \alpha$ which gives two options for continuing the computation. The choice of continuation is under the control of some external agent, which one may wish to model as being cooperative (*angelic*), antagonistic (*demonic*), or *neutral*.

Example 3 (Probabilistic choice). Again there is a single binary choice operator $\mathit{p\text{-}or} : \alpha^2 \to \alpha$ which gives two options for continuing the computation. In this case, the choice of continuation is probabilistic, with a $\frac{1}{2}$ probability of either option being chosen. Other weighted probabilistic choices can be programmed in terms of this fair choice operation.

Example 4 (Global store). We take a set of locations L for storing natural numbers. For each $l \in L$ we have $\mathit{lookup}_l : \alpha^{\mathbf{N}} \to \alpha$ and $\mathit{update}_l : \mathbf{N} \times \alpha \to \alpha$. The computation $\mathit{lookup}_l(V)$ looks up the number at location l and passes it as an argument to the function V, and $\mathit{update}_l(\overline{n}; M)$ stores n at l and then continues with the computation M.

Example 5 (Input/output). Here we have two operators, $\mathit{read} : \alpha^{\mathbf{N}} \to \alpha$ which reads a number from an input channel and passes it as the argument to a function, and $\mathit{write} : \mathbf{N} \times \alpha \to \alpha$ which outputs a number (the first argument) and then continues as the computation given as the second argument.

We next present an operational semantics for our language, under which a computation term evaluates to an *effect tree*: essentially, a coinductively generated term using operations from Σ, and with values and $\bot$ (nontermination) as the generators. This idea appears in [8,21], and our technical treatment follows approach of the latter, adapted to call-by-value.

We define a single-step reduction relation $\rightarrowtail$ between configurations (S, M) consisting of a stack S and a computation M. The computation M is the term under current evaluation. The stack S represents a continuation computation awaiting the termination of M. First, we define a stack-independent reduction relation on computation terms that do not involve **let** at the top level.

$$\begin{array}{l}
(\lambda x:\tau.M)V \;\rightsquigarrow\; M[V/x] \\
\textbf{case}\ Z\ \textbf{of}\ \{Z \Rightarrow M_1; S(x) \Rightarrow M_2\} \;\rightsquigarrow\; M_1 \\
\textbf{case}\ S(V)\ \textbf{of}\ \{Z \Rightarrow M_1; S(x) \Rightarrow M_2\} \quad\rightsquigarrow\quad M_2[V/x] \\
\textbf{fix}(F) \quad\rightsquigarrow\quad \textbf{return}\ \lambda x:\tau.\ \textbf{let}\ F(\lambda y:\tau.\textbf{let fix}\ F \Rightarrow z\ \textbf{in}\ zy) \Rightarrow w\ \textbf{in}\ wx
\end{array}$$

The behaviour of **let** is implemented using a system of stacks where:

Stacks $S ::= \ id \mid S \circ (\ \textbf{let}\ (-) \Rightarrow x\ \textbf{in}\ M)$

We write $S\{N\}$ for the computation term obtained by 'applying' the stack S to N, defined by:

$$\begin{array}{rcl}
id\,\{N\} &=& N \\
(S \circ (\textbf{let}\ (-) \Rightarrow x\ \textbf{in}\ M))\,\{N\} &=& S\{\ \textbf{let}\ N \Rightarrow x\ \textbf{in}\ M\}
\end{array}$$

We write $\mathit{Stack}(\tau, \rho)$ for the set of stacks S such that for any $N \in \mathit{Com}(\tau)$, it holds that $S\{N\}$ is a well-typed expression of type ρ. We define a reduction relation on pairs $\mathit{Stack}(\tau, \rho) \times \mathit{Com}(\tau)$ (denoted $(S_1, M_1) \rightarrowtail (S_2, M_2)$) by:

$$\begin{array}{ll}
(S,\ \textbf{let}\ N \Rightarrow x\ \textbf{in}\ M) \;\rightarrowtail\; (S \circ (\ \textbf{let}\ (-) \Rightarrow x\ \textbf{in}\ M), N) & \\
(S, R) \;\rightarrowtail\; (S, R') & \text{if } R \rightsquigarrow R' \\
(S \circ (\ \textbf{let}\ (-) \Rightarrow x\ \textbf{in}\ M), \textbf{return}\ V) \;\rightarrowtail\; (S, M[V/x]) &
\end{array}$$

We define the notion of *effect tree* for an arbitrary set X, where X is thought of as a set of abstract 'values'.

Definition 1. An *effect tree* (henceforth *tree*), over a set X, determined by a signature Σ of effect operations, is a labelled and possibly infinite tree whose nodes have the possible forms.

1. A leaf node labelled with $\bot$ (the symbol for nontermination).
2. A leaf node labelled with x where $x \in X$.
3. A node labelled σ with children $t_0, \ldots, t_{n-1}$, when $\sigma \in \Sigma$ has arity $\alpha^n \to \alpha$.
4. A node labelled σ with children $t_0, t_1, \ldots$, when $\sigma \in \Sigma$ has arity $\alpha^{\mathbf{N}} \to \alpha$.
5. A node labelled σ_m where $m \in \mathbb{N}$ with children $t_0, \ldots, t_{n-1}$, when $\sigma \in \Sigma$ has arity $\mathbf{N} \times \alpha^n \to \alpha$.
6. A node labelled σ_m where $m \in \mathbb{N}$ with children $t_0, t_1, \ldots$, when $\sigma \in \Sigma$ has arity $\mathbf{N} \times \alpha^{\mathbf{N}} \to \alpha$.

We write TX for the set of trees over X. We define a partial ordering on TX where $t_1 \leq t_2$, if t_1 can be obtained by replacing subtrees of t_2 by $\bot$. This forms an *ω-complete* partial order, meaning that every ascending sequence $t_1 \leq t_2 \leq \dots$ has a least upper bound $\bigsqcup_n t_n$. Let $\mathit{Tree}(\tau) := T\,\mathit{Val}(\tau)$, we will define a reduction relation from computations to trees of values.

Given $f : X \to Y$ and a tree $t \in TX$, we write $t[x \mapsto f(x)] \in TY$ for the tree whose leaves $x \in X$ are renamed to $f(x)$. We have a function $\mu : TTX \to TX$, which takes a tree r of trees and flattens it to a tree $\mu r \in TX$, by taking the labelling tree at each non-$\bot$ leaf of r as the subtree at the corresponding node in μr. The function μ is the multiplication associated with the monad structure of the T operation. The unit of the monad is the map $\eta : X \to TX$ which takes an element $x \in X$ and returns a leaf labelled x.

The operational mapping from a computation $M \in \mathit{Com}(\tau)$ to an effect tree is defined intuitively as follows. Start evaluating the M in the empty stack id, until the evaluation process (which is deterministic) terminates (if this never happens the tree is $\bot$). If the evaluation process terminates at a configuration of the form $(\mathit{id}, \textbf{return } V)$ then the tree is the leaf V. Otherwise the evaluation process can only terminate at a configuration of the form $(S, \sigma(\dots))$ for some effect operation $\sigma \in \Sigma$. In this case, create an internal node in the tree of the appropriate kind (depending on σ) and continue generating each child tree of this node by repeating the above process by evaluating an appropriate continuation computation, starting from a configuration with the current stack S.

The following (somewhat technical) definition formalises the idea outlined above in a mathematically concise way. We define a family of maps $|-,-|_{(-)} : \mathit{Stack}(\tau, \rho) \times \mathit{Com}(\tau) \times \mathbb{N} \to \mathit{Tree}(\rho)$ indexed over τ, and ρ by:

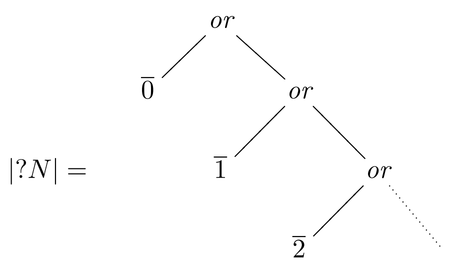
$$|S, M|_0 = \bot$$

$$|S, M|_{n+1} = \begin{cases} V & \text{if } S = \mathit{id} \wedge M = \textbf{return } V \\ |S', M'|_n & \text{if } (S, M) \rightarrowtail (S', M') \\ \sigma(|S, M_0|_n, \dots, |S, M_{m-1}|_n) & \sigma : \alpha^m \to \alpha, M = \sigma(M_0, \dots, M_{m-1}) \\ \sigma(|S, V\overline{0}|_n, |S, V\overline{1}|_n, \dots) & \sigma : \alpha^{\mathbf{N}} \to \alpha, M = \sigma(V) \\ \sigma_k(|S, M_0|_n, \dots, |S, M_{m-1}|_n) & \sigma : \mathbf{N} \times \alpha^m \to \alpha, M = \sigma(\overline{k}, M_0, \dots, M_{m-1}) \\ \sigma_k(|S, V\overline{0}|_n, |S, V\overline{1}|_n, \dots) & \sigma : \mathbf{N} \times \alpha^{\mathbf{N}} \to \alpha, M = \sigma(\overline{k}, V) \\ \bot & \text{otherwise} \end{cases}$$

It follows that $|S, M|_n \leq |S, M|_{n+1}$ in the given ordering on trees. We write $|-|_{(-)} : \mathit{Com}(\tau) \times \mathbb{N} \to \mathit{Tree}(\tau)$ for the function defined by $|M|_n = |\mathit{id}, M|_n$. Using this we can give the operational interpretation of computation terms as effect trees by defining $|-| : \mathit{Com}(\tau) \to \mathit{Tree}(\tau)$ by $|M| := \bigsqcup_n |M|_n$.

Example 3 (Nondeterminism). Nondeterministically generate a natural number:

$$?N := \textbf{ let fix}(\lambda x : \mathbf{1} \to \mathbf{N}.\, \mathit{or}(\lambda y : \mathbf{1}.\, Z,\ \lambda y : \mathbf{1}.\ \textbf{let } xy \Rightarrow z \textbf{ in } S(z))) \Rightarrow w \textbf{ in } w*$$



3 Behavioural Logic and Modalities

The goal of this section is to motivate and formulate a logic for expressing *behavioural properties* of programs. In our language, program means (well-typed) term, and we shall be interested both in properties of *computations* and in properties of *values*. Accordingly, we define a logic that contains both *value formulas* and *computation formulas*. We shall use lower case Greek letters $\phi, \psi, \ldots$ for the former, and upper case Greek letters $\mathit{\Phi}, \mathit{\Psi}, \ldots$ for the latter. Our logic will thus have two satisfaction relations

$$V \models \phi \qquad\qquad M \models \mathit{\Phi}$$

which respectively assert that "value V enjoys the value property expressed by ϕ" and "computation M enjoys the computation property expressed by $\mathit{\Phi}$".

In order to motivate the detailed formulation of the logic, it is useful to identify criteria that will guide the design.

(C1) The logic should express only 'behaviourally meaningful' properties of programs. This guides us to build the logic upon primitive notions that have a direct behavioural interpretation according to a natural understanding of program behaviour.

(C2) The logic should be as expressive as possible within the constraints imposed by criterion (C1).

For every type τ, we define a collection $\mathit{VF}(\tau)$ of *value formulas*, and a collection $\mathit{CF}(\tau)$ of *computation formulas*, as motivated above.

Since boolean logical connectives say nothing themselves about computational behaviour, it is a reasonable general principle that 'behavioural properties' should be closed under such connectives. Thus, in keeping with criterion (C2), which asks for maximal expressivity, we close each set $\mathit{CF}(\tau)$ and $\mathit{VF}(\tau)$, of computation and value formulas, under infinitary propositional logic.

In addition to closure under infinitary propositional logic, each set $\mathit{VF}(\tau)$ contains a collection of *basic* value formulas, from which compound formulas are constructed using (infinitary) propositional connectives.[1] The choice of basic formulas depends on the type τ.

[1] We call such formulas *basic* rather than *atomic* because they include formulas such as $(V \mapsto \mathit{\Phi})$, discussed below, which are built from other formulas.

In the case of the natural numbers type, we include a basic value formula $\{n\} \in \mathit{VF}(\mathbf{N})$, for every $n \in \mathbb{N}$. The semantics of this formula are given by:

$$V \models \{n\} \quad \Leftrightarrow \quad V = \overline{n}.$$

By the closure of $\mathit{VF}(\mathbf{N})$ under infinitary disjunctions, every subset of $\mathbb{N}$ can be represented by some value formula. Moreover, since a general value formula in $\mathit{VF}(\mathbf{N})$ is an infinitary boolean combination of basic formulas of the form $\{n\}$, the value formulas represent exactly the subsets on $\mathbb{N}$.

For the unit type, we do not require any basic value formulas. The unit type has only one value, $*$. The two subsets of this singleton set of values are defined by the formulas $\bot$ ('falsum', given as an empty disjunction), and $\top$ (the truth constant, given as an empty conjunction).

For a function type $\tau \to \rho$, we want each basic formula to express a fundamental behavioural constraint on values (i.e., λ-abstractions) W of type $\tau \to \rho$. In keeping with the applicative nature of functional programming, the only way in which a λ-abstraction can be used to generate behaviour is to apply it to an argument of type τ, which, because we are in a call-by-value setting, must be a value V. The application of W to V results in a computation WV of type ρ, whose properties can be probed using computation formulas in $\mathit{CF}(\rho)$. Based on this, for every value $V \in \mathit{Val}(\tau)$ and computation formula $\Phi \in \mathit{CF}(\rho)$, we include a basic value formula $(V \mapsto \Phi) \in \mathit{VF}(\tau \to \rho)$ with the semantics:

$$W \models (V \mapsto \Phi) \quad \Leftrightarrow \quad WV \models \Phi.$$

Using this simple construct, based on application to a single argument V, other natural mechanisms for expressing properties of λ-abstractions are definable, using infinitary propositional logic. For example, given $\phi \in \mathit{VF}(\tau)$ and $\Psi \in \mathit{CF}(\rho)$, the definition

$$(\phi \mapsto \Psi) \;:=\; \bigwedge\{(V \mapsto \Psi) \mid V \in \mathit{Val}(\tau),\, V \models \phi\} \tag{1}$$

defines a formula whose derived semantics is

$$W \models (\phi \mapsto \Psi) \quad \Leftrightarrow \quad \forall V \in \mathit{Val}(\tau).\ V \models \phi \text{ implies } WV \models \Psi. \tag{2}$$

In Sect. 7, we shall consider the possibility of changing the basic value formulas in $\mathit{VF}(\tau \to \rho)$ to formulas $(\phi \mapsto \Psi)$.

It remains to explain how the basic computation formulas in $\mathit{CF}(\tau)$ are formed. For this we require a given set $\mathcal{O}$ of *modalities*, which depends on the algebraic effects contained in the language. The basic computation formulas in $\mathit{CF}(\tau)$ then have the form $o\,\phi$, where $o \in \mathcal{O}$ is one of the available modalities, and ϕ is a value formula in $\mathit{VF}(\tau)$. Thus a modality 'lifts' properties of values of type τ to properties of computations of type τ.

In order to give semantics to computation formulas $o\,\phi$, we need a general theory of the kind of modality under consideration. This is one of the main contributions of the paper. Before presenting the general theory, we first consider motivating examples, using our running examples of algebraic effects.

Example 0 (Pure functional computation). Define $\mathcal{O} = \{\downarrow\}$. Here the single modality $\downarrow$ is the *termination modality*: $\downarrow\phi$ asserts that a computation terminates with a return value V satisfying ϕ. This is formalised using effect trees:

$$M \models \downarrow\phi \quad \Leftrightarrow \quad |M| \text{ is a leaf } V \text{ and } V \models \phi.$$

Note that, in the case of pure functional computation, all trees are leaves: either value leaves V, or nontermination leaves $\perp$.

Example 1 (Error). Define $\mathcal{O} = \{\downarrow\} \cup \{\mathsf{E}_e \mid e \in E\}$. The semantics of the termination modality $\downarrow$ is defined as above. The *error modality* E_e flags error e:

$$M \models \mathsf{E}_e\phi \quad \Leftrightarrow \quad |M| \text{ is a node labelled with } \mathit{raise}_e.$$

(Because raise_e is an operation of arity 0, a raise_e node in a tree has 0 children.) Note that the semantics of $\mathsf{E}_e\phi$ makes no reference to ϕ. Indeed it would be natural to consider E_e as a basic computation formula in its own right, which could be done by introducing a notion of 0-argument modality, and considering E_e as such. In this paper, however, we keep the treatment uniform by always considering modalities as unary operations, with natural 0-argument modalities subsumed as unary modalities with redundant argument.

Example 2 (Nondeterminism). Define $\mathcal{O} = \{\Diamond, \Box\}$ with:

$$M \models \Diamond\phi \quad \Leftrightarrow \quad |M| \text{ has some leaf } V \text{ such that } V \models \phi$$
$$M \models \Box\phi \quad \Leftrightarrow \quad |M| \text{ has finite height and every leaf is a value } V \text{ s.t. } V \models \phi.$$

Including both modalities amounts to a neutral view of nondeterminism. In the case of angelic nondeterminism, one would include just the $\Diamond$ modality; in that of demonic nondeterminism, just the $\Box$ modality. Because of the way the semantic definitions interact with termination, the modalities $\Box$ and $\Diamond$ are not De Morgan duals. Indeed, each of the three possibilities $\{\Diamond, \Box\}, \{\Diamond\}, \{\Box\}$ for $\mathcal{O}$ leads to a logic with a different expressivity.

Example 3 (Probabilistic choice). Define $\mathcal{O} = \{\mathsf{P}_{>q} \mid q \in \mathbb{Q}, 0 \le q < 1\}$ with:

$$M \models \mathsf{P}_{>q}\,\phi \quad \Leftrightarrow \quad \mathbf{P}(|M| \text{ terminates with a value in } \{V \mid V \models \phi\}) > q,$$

where the probability on the right is the probability that a run through the tree $|M|$, starting at the root, and making an independent fair probabilistic choice at each branching node, terminates at a value node with a value V in the set $\{V \mid V \models \phi\}$. We observe that the restriction to rational thresholds q is immaterial, as, for any real r with $0 \le r < 1$, we can define:

$$\mathsf{P}_{>r}\,\phi \;:=\; \bigvee\{\mathsf{P}_{>q}\,\phi \mid q \in \mathbb{Q},\, r < q < 1\}.$$

Similarly, we can define non-strict threshold modalities, for $0 < r \le 1$, by:

$$\mathsf{P}_{\ge r}\,\phi \;:=\; \bigwedge\{\mathsf{P}_{>q}\,\phi \mid q \in \mathbb{Q},\, 0 \le q < r\}.$$

Also, we can exploit negation to define modalities expressing strict and non-strict upper bounds on probabilities. Notwithstanding the definability of non-strict and upper-bound thresholds, we shall see later that it is important that we include only strict lower-bound modalities in our set $\mathcal{O}$ of primitive modalities.

Example 4 (Global store). For a set of locations L, define the set of states by $\mathit{State} = \mathbb{N}^L$. The modalities are $\mathcal{O} = \{(s \rightarrowtail r) \mid s, r \in \mathit{State}\}$, where informally:

$$M \models (s \rightarrowtail r)\,\phi \quad \Leftrightarrow \quad \text{the execution of } M,\ \text{starting in state } s,\ \text{terminates in final state } r \text{ with return value } V \text{ such that } V \models \phi.$$

We make the above definition precise using the effect tree of M. Define

$$\mathit{exec} : TX \times \mathit{State} \to X \times \mathit{State},$$

for any set X, to be the least partial function satisfying:

$$\mathit{exec}(t, s) = \begin{cases} (x, s) & \text{if } t \text{ is a leaf labelled with } x \in X \\ \mathit{exec}(t_{s(l)}, s) & \text{if } t = \mathit{lookup}_l(t_0, t_1, \cdots) \text{ and } \mathit{exec}(t_{s(l)}, s) \text{is defined} \\ \mathit{exec}(t', s[l := n]) & \text{if } t = \mathit{update}_{l,n}(t') \text{ and } \mathit{exec}(t', s[l := n]) \text{ is defined,} \end{cases}$$

where $s[l := n]$ is the evident modification of state s. Intuitively, $\mathit{exec}(t, s)$ defines the result of "executing" the tree of commands in effect tree t starting in state s, whenever this execution terminates. In terms of operational semantics, it can be viewed as defining a 'big-step' semantics for effect trees (in the signature of global store). We can now define the semantics of the $(s \rightarrowtail r)$ modality formally:

$$M \models (s \rightarrowtail r)\,\phi \quad \Leftrightarrow \quad \mathit{exec}(|M|, s) = (V, r) \text{ where } V \models \phi.$$

Example 5 (Input/output). Define an *i/o-trace* to be a word w over the alphabet

$$\{?n \mid n \in \mathbb{N}\} \cup \{!n \mid n \in \mathbb{N}\}.$$

The idea is that such a word represents an input/output sequence, where $?n$ means the number n is given in response to an input prompt, and $!n$ means that the program outputs n. Define the set of modalities

$$\mathcal{O} = \{\langle w \rangle{\downarrow},\ \langle w \rangle_{\ldots} \mid w \text{ an i/o-trace}\}.$$

The intuitive semantics of these modalities is as follows.

$$\begin{aligned} M \models \langle w \rangle{\downarrow}\,\phi \quad &\Leftrightarrow \quad w \text{ is a complete i/o-trace for the execution of } M \\ &\qquad\quad \text{resulting in termination with } V \text{ s.t. } V \models \phi \\ M \models \langle w \rangle_{\ldots}\,\phi \quad &\Leftrightarrow \quad w \text{ is an initial i/o-trace for the execution of } M. \end{aligned}$$

In order to define the semantics of formulas precisely, we first define relations $t \models \langle w \rangle{\downarrow}\,P$ and $t \models \langle w \rangle_{\ldots}$, between $t \in TX$ and $P \subseteq X$, by induction on words

$$\frac{n \in \mathbb{N}}{\{n\} \in \mathit{VF}(\mathbf{N})}(1) \qquad \frac{V : \tau \qquad \Phi \in \mathit{CF}(\rho)}{(V \mapsto \Phi) \in \mathit{VF}(\tau \to \rho)}(2) \qquad \frac{\phi \in \mathit{VF}(\tau) \qquad o \in \mathcal{O}}{o\,\phi \in \mathit{CF}(\tau)}(3)$$

$$\frac{\phi : I \to \mathit{VF}(\tau)}{\bigvee_I \phi \in \mathit{VF}(\tau)}(4) \qquad \frac{\phi : I \to \mathit{VF}(\tau)}{\bigwedge_I \phi \in \mathit{VF}(\tau)}(5) \qquad \frac{\phi \in \mathit{VF}(\tau)}{\neg\phi \in \mathit{VF}(\tau)}(6)$$

$$\frac{\Phi : I \to \mathit{CF}(\tau)}{\bigvee_I \Phi \in \mathit{CF}(\tau)}(7) \qquad \frac{\Phi : I \to \mathit{CF}(\tau)}{\bigwedge_I \Phi \in \mathit{CF}(\tau)}(8) \qquad \frac{\Phi \in \mathit{CF}(\tau)}{\neg\Phi \in \mathit{CF}(\tau)}(9)$$

Fig. 2. The logic $\mathcal{V}$

(Note that we are overloading the $\models$ symbol.) In the following, we write ε for the empty word, and we use textual juxtaposition for concatenation of words.

$$\begin{aligned}
t &\models \langle\varepsilon\rangle{\downarrow} P &\Leftrightarrow&\quad t \text{ is a leaf } x \text{ and } x \in P\\
t &\models \langle(?n)\,w\rangle{\downarrow} P &\Leftrightarrow&\quad t = \mathit{read}(t_0, t_1, \dots) \text{ and } t_n \models \langle w\rangle{\downarrow} P\\
t &\models \langle(!n)\,w\rangle{\downarrow} P &\Leftrightarrow&\quad t = \mathit{write}_n(t') \text{ and } t' \models \langle w\rangle{\downarrow} P\\
t &\models \langle\varepsilon\rangle_{\dots} &\Leftrightarrow&\quad \text{true}\\
t &\models \langle(?n)\,w\rangle_{\dots} &\Leftrightarrow&\quad t = \mathit{read}(t_0, t_1, \dots) \text{ and } t_n \models \langle w\rangle_{\dots}\\
t &\models \langle(!n)\,w\rangle_{\dots} &\Leftrightarrow&\quad t = \mathit{write}_n(t') \text{ and } t' \models \langle w\rangle_{\dots}
\end{aligned}$$

The formal semantics of modalities is now easily defined by:

$$\begin{aligned}
M &\models \langle w\rangle{\downarrow}\,\phi &\Leftrightarrow&\quad |M| \models \langle w\rangle{\downarrow}\,\{V \mid V \models \phi\}\\
M &\models \langle w\rangle_{\dots}\,\phi &\Leftrightarrow&\quad |M| \models \langle w\rangle_{\dots}.
\end{aligned}$$

Note that, as in Example 1, the formula argument of the $\langle w\rangle_{\dots}$ modality is redundant. Also, note that our modalities for input/output could naturally be formed by combining the termination modality $\downarrow$, which lifts value formulas to computation formulas, with sequences of atomic modalities $\langle ?n\rangle$ and $\langle !n\rangle$ acting directly on computation formulas. In this paper, we do not include such modalities, acting on computation formulas, in our general theory. But this is a natural avenue for future consideration.

We now give a formal treatment of the logic and its semantics, in full generality. We assume given a signature Σ of effect operations, as in Sect. 2. And we assume given a set $\mathcal{O}$, whose elements we call *modalities*.

We call our main behavioural logic $\mathcal{V}$, where the letter $\mathcal{V}$ is chosen as a reference to the fact that the basic formula at function type specifies function behaviour on individual value arguments V.

Definition 2 (The logic $\mathcal{V}$). The classes $\mathit{VF}(\tau)$ and $\mathit{CF}(\tau)$ of *value* and *computation formulas*, for each type τ, are mutually inductively defined by the rules in Fig. 2. In this, I can be instantiated to any set, allowing for arbitrary conjunctions and disjunctions. When I is $\emptyset$, we get the special formulas $\top = \bigwedge_\emptyset$ and $\bot = \bigvee_\emptyset$. The use of arbitrary index sets means that formulas, as defined, form a proper class. However, we shall see below that countable index sets suffice.

In order to specify the semantics of modal formulas, we require a connection between modalities and effect trees, which is given by an interpretation function

$$[\![\cdot]\!] : \mathcal{O} \to \mathcal{P}(T\mathbf{1}).$$

That is, every modality $o \in \mathcal{O}$ is mapped to a subset $[\![o]\!] \subseteq T\mathbf{1}$ of unit-type effect trees. Given a subset $P \subseteq X$ (e.g. given by a formula) and a tree $t \in TX$ we can define a unit-type tree $t[\in P] \in T\mathbf{1}$ as the tree created by replacing the leaves of t that belong to P by $*$ and the others by $\bot$. In the case that P is the subset $\{V \mid V \models \phi\}$ specified by a formula $\phi \in \mathit{VF}(\tau)$, we also write $t[\models \phi]$ for $t[\in P]$.

We can now formally define the two satisfaction relations $\models \subseteq \mathit{Val}(\tau) \times \mathit{VF}(\tau)$ and $\models \subseteq \mathit{Com}(\tau) \times \mathit{CF}(\tau)$, mutually inductively, by:

$$\begin{array}{rcl} \overline{m} \models \{n\} & \Leftrightarrow & m = n \\ W \models (V \mapsto \Phi) & \Leftrightarrow & WV \models \Phi \\ M \models o\,\phi & \Leftrightarrow & |M|\,[\models \phi] \in [\![o]\!] \\ W \models \neg\phi & \Leftrightarrow & \neg(W \models \phi). \end{array}$$

We omit the evident clauses for the other propositional connectives. We remark that all conjunctions and disjunctions are semantically equivalent to countable ones, because value and computation formulas are interpreted over sets of terms, $\mathit{Val}(\tau)$ and $\mathit{Com}(\tau)$, which are countable.

We end this section by revisiting our running examples, and showing, in each case, that the example modalities presented above are all specified by suitable interpretation functions $[\![\cdot]\!] : \mathcal{O} \to \mathcal{P}(T\mathbf{1})$.

Example 0 (Pure functional computation). We have $\mathcal{O} = \{\downarrow\}$. Define:

$$[\![\downarrow]\!] = \{\, * \,\} \quad \text{(where } * \text{ is the tree with single node } *)$$

Example 1 (Error). We have $\mathcal{O} = \{\downarrow\} \cup \{\mathsf{E}_e \mid e \in E\}$. Define:

$$[\![\mathsf{E}_e]\!] = \{\, \mathit{raise}_e \,\}.$$

Example 2 (Nondeterminism). We have $\mathcal{O} = \{\Diamond, \Box\}$. Define:

$$\begin{array}{rcl} [\![\Diamond]\!] & = & \{t \mid t \text{ has some * leaf}\} \\ [\![\Box]\!] & = & \{t \mid t \text{ has finite height and every leaf is a *}\}. \end{array}$$

Example 3 (Probabilistic choice). $\mathcal{O} = \{\mathsf{P}_{>q} \mid q \in \mathbb{Q},\, 0 \leq q < 1\}$. Define:

$$[\![\mathsf{P}_{>q}]\!] = \{t \mid \mathbf{P}(\, t \text{ terminates with a * leaf}\,) > q\}.$$

Example 4 (Global store). $\mathcal{O} = \{(s \rightarrowtail r) \mid s, r \in \mathit{State}\}$. Define:

$$[\![(s \rightarrowtail r)]\!] = \{t \mid \mathit{exec}(t, s) = (*, r)\}.$$

Example 5 (Input/output). $\mathcal{O} = \{\langle w \rangle\!\downarrow,\, \langle w \rangle_{\dots} \mid w \text{ an i/o-trace}\}$. Define:

$$\begin{array}{rcl} [\![\langle w \rangle\!\downarrow]\!] & = & \{t \mid t \models \langle w \rangle\!\downarrow \{*\}\,\} \\ [\![\langle w \rangle_{\dots}]\!] & = & \{t \mid t \models \langle w \rangle_{\dots}\}. \end{array}$$

4 Behavioural Equivalence

The goal of this section is to precisely formulate our main theorem: under suitable conditions, the behavioural equivalence determined by the logic $\mathcal{V}$ of Sect. 3 is a congruence. In order to achieve this, it will be useful to consider the *positive fragment* $\mathcal{V}^+$ of $\mathcal{V}$.

Definition 3 (The logic $\mathcal{V}^+$). The logic $\mathcal{V}^+$ is the fragment of $\mathcal{V}$ consisting of those formulas in $\mathit{VF}(\tau)$ and $\mathit{CF}(\tau)$ that do not contain negation.

Whenever we have a logic $\mathcal{L}$ whose value and computation formulas are given as subcollections $\mathit{VF}_{\mathcal{L}}(\tau) \subseteq \mathit{VF}(\tau)$ and $\mathit{CF}_{\mathcal{L}}(\tau) \subseteq \mathit{CF}(\tau)$, then $\mathcal{L}$ determines a preorder (and hence also an equivalence relation) between terms of the same type and aspect.

Definition 4 (Logical preorder and equivalence). Given a fragment $\mathcal{L}$ of $\mathcal{V}$, we define the *logical preorder* $\sqsubseteq_{\mathcal{L}}$, between well-typed terms of the same type and aspect, by:

$$\begin{aligned} V \sqsubseteq_{\mathcal{L}} W \quad &\Leftrightarrow \quad \forall \phi \in \mathit{VF}_{\mathcal{L}}(\tau),\ V \models \phi \Rightarrow W \models \phi \\ M \sqsubseteq_{\mathcal{L}} N \quad &\Leftrightarrow \quad \forall \Phi \in \mathit{CF}_{\mathcal{L}}(\tau),\ M \models \Phi \Rightarrow N \models \Phi \end{aligned}$$

The *logical equivalence* $\equiv_{\mathcal{L}}$ on terms is the equivalence relation induced by the preorder (the intersection of $\sqsubseteq_{\mathcal{L}}$ and its converse).

In the case that formulas in $\mathcal{L}$ are closed under negation, it is trivial that the preorder $\sqsubseteq_{\mathcal{L}}$ is already an equivalence relation, and hence coincides with $\equiv_{\mathcal{L}}$. Thus we shall only refer specifically to the preorder $\sqsubseteq_{\mathcal{L}}$, for fragments, such as $\mathcal{V}^+$, that are not closed under negation.

The two main relations of interest to us in this paper are the primary relations determined by $\mathcal{V}$ and $\mathcal{V}^+$: full *behavioural equivalence* $\equiv_{\mathcal{V}}$; and the *positive behavioural preorder* $\sqsubseteq_{\mathcal{V}^+}$ (which induces *positive behavioural equivalence* $\equiv_{\mathcal{V}^+}$).

We next formulate the appropriate notion of (pre)congruence to apply to the relations $\equiv_{\mathcal{V}}$ and $\sqsubseteq_{\mathcal{V}^+}$. These two preorders are examples of *well-typed relations* on closed terms. Any such relation can be extended to a relation on open terms in the following way. Given a well-typed relation $\mathcal{R}$ on closed terms, we define the *open extension* $\mathcal{R}^\circ$ where $\Gamma \vdash M \mathcal{R}^\circ N : \tau$ precisely when, for every well-typed vector of closed values $\overrightarrow{V} : \Gamma$, it holds that $M[\overrightarrow{V}]\, \mathcal{R}\, N[\overrightarrow{V}]$. The correct notion of precongruence for a well-typed preorder on closed terms, is to ask for its open extension to be *compatible* in the sense of the definition below; see, e.g., [10,19] for further explanation.

Definition 5 (Compatibility). A well-typed open relation $\mathcal{R}$ is said to be *compatible* if it is closed under the rules in Fig. 3.

We now state our main congruence result, although we have not yet defined the conditions it depends upon.

$$\frac{}{\Gamma, x : \tau \vdash x \mathrel{\mathcal{R}} x : \tau} \qquad \frac{}{\Gamma \vdash Z \mathrel{\mathcal{R}} Z : \mathbf{N}} \qquad \frac{\Gamma \vdash V \mathrel{\mathcal{R}} V' : \mathbf{N}}{\Gamma \vdash S(V) \mathrel{\mathcal{R}} S(V') : \mathbf{N}}$$

$$\frac{\Gamma \vdash V \mathrel{\mathcal{R}} V' : \tau}{\Gamma \vdash \mathbf{return}(V) \mathrel{\mathcal{R}} \mathbf{return}(V') : \tau} \qquad \frac{\Gamma, x : \tau \vdash M \mathrel{\mathcal{R}} M' : \rho}{\Gamma \vdash (\lambda x : \tau.M) \mathrel{\mathcal{R}} (\lambda x : \tau.M') : \tau \to \rho}$$

$$\frac{\Gamma \vdash V \mathrel{\mathcal{R}} V' : \tau \to \rho \qquad \Gamma \vdash W \mathrel{\mathcal{R}} W' : \tau}{\Gamma \vdash (VW) \mathrel{\mathcal{R}} (V'W') : \rho} \qquad \frac{\Gamma \vdash V \mathrel{\mathcal{R}} V' : (\tau \to \rho) \to (\tau \to \rho)}{\Gamma \vdash \mathbf{fix}(V) \mathrel{\mathcal{R}} \mathbf{fix}(V') : \tau \to \rho}$$

$$\frac{\Gamma \vdash V \mathrel{\mathcal{R}} V' : \mathbf{N} \qquad \Gamma \vdash M \mathrel{\mathcal{R}} M' : \tau \qquad \Gamma, x : \mathbf{N} \vdash N \mathrel{\mathcal{R}} N' : \tau}{\Gamma \vdash \ \mathbf{case}\ V\ \mathbf{of}\ \{Z \Rightarrow M; S(x) \Rightarrow N\} \mathrel{\mathcal{R}}\ \mathbf{case}\ V'\ \mathbf{of}\ \{Z \Rightarrow M'; S(x) \Rightarrow N'\} : \tau}$$

$$\frac{\Gamma \vdash M \mathrel{\mathcal{R}} M' : \tau \qquad \Gamma, x : \tau \vdash N \mathrel{\mathcal{R}} N' : \rho}{\Gamma \vdash \mathbf{let}\ M \Rightarrow x\ \mathbf{in}\ N \mathrel{\mathcal{R}} \mathbf{let}\ M' \Rightarrow x\ \mathbf{in}\ N' : \rho}$$

$$\frac{\Gamma \vdash M_i \mathrel{\mathcal{R}} M_i' : \tau}{\Gamma \vdash \sigma(M_0, M_1, ...) \mathrel{\mathcal{R}} \sigma(M_0', M_1', ...) : \tau} \quad \frac{\Gamma \vdash V \mathrel{\mathcal{R}} V' : \mathbf{N} \qquad \Gamma \vdash M_i \mathrel{\mathcal{R}} M_i' : \tau}{\Gamma \vdash \sigma(V; M_0, M_1, ...) \mathrel{\mathcal{R}} \sigma(V'; M_0', M_1', ...) : \tau}$$

$$\frac{\Gamma \vdash V \mathrel{\mathcal{R}} V' : \mathbf{N} \to \tau}{\Gamma \vdash \sigma(V) \mathrel{\mathcal{R}} \sigma(V') : \tau} \qquad \frac{\Gamma \vdash V \mathrel{\mathcal{R}} V' : \mathbf{N} \qquad \Gamma \vdash W \mathrel{\mathcal{R}} W' : \mathbf{N} \to \tau}{\Gamma \vdash \sigma(V; W) \mathrel{\mathcal{R}} \sigma(V'; W') : \tau}$$

Fig. 3. Rules for compatibility

Theorem 1. *If $\mathcal{O}$ is a decomposable set of Scott-open modalities then the open extensions of $\equiv_{\mathcal{V}}$ and $\sqsubseteq_{\mathcal{V}^+}$ are both compatible. (It is an immediate consequence that the open extension of $\equiv_{\mathcal{V}^+}$ is also compatible.)*

The Scott-openness condition refers to the *Scott topology* on $T\mathbf{1}$.

Definition 6. We say that $o \in \mathcal{O}$ is *upwards closed* if $[\![o]\!]$ is an upper-closed subset of $T\mathbf{1}$; i.e., if $t \in [\![o]\!]$ implies $t' \in [\![o]\!]$ whenever $t \leq t'$.

Definition 7. We say that $o \in \mathcal{O}$ is *Scott-open* if $[\![o]\!]$ is an open subset in the Scott topology on $T\mathbf{1}$; i.e., $[\![o]\!]$ is upper closed and, whenever $t_1 \leq t_2 \leq \dots$ is an ascending chain in $T\mathbf{1}$ with supremum $\sqcup_i t_i \in [\![o]\!]$, we have $t_n \in [\![o]\!]$ for some n.

Before formulating the property of *decomposability*, we make some simple observations about the positive preorder $\sqsubseteq_{\mathcal{V}^+}$.

Lemma 8. *For any $V_0, V_1 \in Val(\rho \to \tau)$, we have $V_0 \sqsubseteq_{\mathcal{V}^+} V_1$ if and only if:*

$$\forall W \in Val(\rho), \forall \Psi \in CF_{\mathcal{V}^+}(\tau),\ V_0 \ \models\ (W \mapsto \Psi) \textit{ implies } V_1 \ \models\ (W \mapsto \Psi).$$

Lemma 9. *For any $M_0, M_1 \in Com(\tau)$, we have $M_0 \sqsubseteq_{\mathcal{V}^+} M_1$ if and only if:*

$$\forall o \in \mathcal{O}, \forall \phi \in VF_{\mathcal{V}^+}(\tau),\ M_0 \ \models\ o\,\phi \textit{ implies } M_1 \ \models\ o\,\phi.$$

Similar characterisations, with appropriate adjustments, hold for behavioural equivalence $\equiv_{\mathcal{V}}$.

The decomposability property is formulated using an extension of the positive preorder $\sqsubseteq_{\mathcal{V}^+}$, at unit type, from a relation on computations to a relation on arbitrary effect trees. Accordingly, we define a preorder $\preceq$ on $T\mathbf{1}$ by:

$$t \preceq t' \quad \Leftrightarrow \quad \forall o \in \mathcal{O},\ (t \in [\![o]\!] \Rightarrow t' \in [\![o]\!]) \wedge (t[\in \emptyset] \in [\![o]\!] \Rightarrow t'[\in \emptyset] \in [\![o]\!]).$$

Proposition 10. *For computations $M, N \in Com(\mathbf{1})$, it holds that $|M| \preceq |N|$ if and only if $M \sqsubseteq_{\mathcal{V}^+} N$.*

Proof. The defining condition for $|M| \preceq |N|$ unwinds to:

$$\forall o \in \mathcal{O},\ (M \ \models\ o\top \text{ implies } N \models\ o\top) \wedge (M \ \models\ o\bot \text{ implies } N \ \models\ o\bot).$$

This coincides with $M \sqsubseteq_{\mathcal{V}^+} N$ by Lemma 9. □

We now formulate the required notion of decomposability. We first give the general definition, and then follow it with a related notion of *strong decomposability*, which can be more convenient to establish in examples. Both definitions are unavoidably technical in nature.

For any relation $\mathcal{R} \subseteq X \times Y$ and subset $A \subseteq X$, we write $\mathcal{R}^{\uparrow}A$ for the right set $\{y \in Y \mid \exists x \in A, x\mathcal{R}y\}$. This allows use to easily define our required notion.

Definition 11 (Decomposability). We say that $\mathcal{O}$ is *decomposable* if, for all $r, r' \in TT\mathbf{1}$, we have:

$$(\forall A \subseteq T\mathbf{1},\ r[\in A] \preceq r'[\in \preceq^{\uparrow}A]) \quad \Rightarrow \quad \mu r \preceq \mu r'.$$

Corollary 22 in Sect. 5, may help to motivate the formulation of the above property, which might otherwise appear purely technical. The following stronger version of decomposability, which suffices for all examples considered in the paper, is perhaps easier to understand in its own right.

Definition 12 (Strong decomposability). We say that $\mathcal{O}$ is *strongly decomposable* if, for every $r \in TT\mathbf{1}$ and $o \in \mathcal{O}$ for which $\mu r \in [\![o]\!]$, there exists a collection $\{(o_i, o'_i)\}_{i\in I}$ of pairs of modalities such that:

1. $\forall i \in I,\ r[\in [\![o'_i]\!]] \in [\![o_i]\!]$; and
2. for every $r' \in TT\mathbf{1}$, $(\forall i \in I,\ r'[\in [\![o'_i]\!]] \in [\![o_i]\!]\,)$ implies $\mu r' \in [\![o]\!]$.

Proposition 13. *If $\mathcal{O}$ is a strongly decomposable then it is decomposable.*

Proof. Suppose that $r[\in A] \preceq r'[\in (\preceq^{\uparrow} A)]$ holds for every $A \subseteq T\mathbf{1}$. Assume that $\mu r \in [\![o]\!] \in \mathcal{O}$. Then strong decomposability gives a collection $\{(o_i, o'_i)\}_I$. By the definition of $\preceq$, for each o'_i we have $\preceq^{\uparrow} [\![o'_i]\!] = [\![o'_i]\!]$. By the initial assumption, $r[\in [\![o'_i]\!]] \in [\![o_i]\!]$ implies $r'[\in (\preceq^{\uparrow} [\![o'_i]\!])] \in [\![o_i]\!]$, and hence $r'[\in [\![o'_i]\!]] \in [\![o_i]\!]$. This holds for every i, so by strong decomposability $\mu r' \in [\![o]\!]$. We have shown that $\mu r \in [\![o]\!]$ implies $\mu r' \in [\![o]\!]$. One can prove similarly that $\mu r[\in \emptyset] \in [\![o]\!]$ implies that $\mu r'[\in \emptyset] \in [\![o]\!]$ by observing that $\preceq^{\uparrow} \{x \mid x[\in \emptyset] \in [\![o'_i]\!]\} = \{x \mid x[\in \emptyset] \in [\![o'_i]\!]\}$. Thus it holds that $\mu r \preceq \mu r'$ and hence $\mathcal{O}$ is decomposable. □

We end this section by again looking at our running examples, and showing, in each case, that the identified collection $\mathcal{O}$ of modalities is Scott-closed (hence upwards closed) and strongly decomposable (hence decomposable). For any of the examples, upwards closure is easily established, so we will not show it here.

Example 0 (Pure functional computation). We have $\mathcal{O} = \{\downarrow\}$ and $[\![\downarrow]\!] = \{*\}$. Scott openness holds since if $\sqcup_i t_i = *$ then for some i we must already have $t_i = *$. It is strongly decomposable since: $\mu r \in [\![\downarrow]\!] \Leftrightarrow r[\in [\![\downarrow]\!]] \in [\![\downarrow]\!]$, which means r returns a tree t which is a leaf $*$.

Example 1 (Error). We have $\mathcal{O} = \{\downarrow\} \cup \{\mathsf{E}_e \mid e \in E\}$ and $[\![\mathsf{E}_e]\!] = \{\, \mathit{raise}_e \,\}$. Scott-openness holds for both modalities for the same reason as in the previous example, and its strongly decomposable since:

$$\mu r \in [\![\downarrow]\!] \quad \Leftrightarrow \quad r[\in [\![\downarrow]\!]] \in [\![\downarrow]\!].$$

Which means r returns a tree t which returns $*$.

$$\mu r \in [\![\mathsf{E}_e]\!] \quad \Leftrightarrow \quad r[\in [\![\mathsf{E}_e]\!]] \in [\![\mathsf{E}_e]\!] \vee r[\in [\![\mathsf{E}_e]\!]] \in [\![\downarrow]\!].$$

Which means r raises an error, or returns a tree that raises an error.

Example 2 (Nondeterminism). We have $\mathcal{O} = \{\Diamond, \Box\}$. The Scott-openness of $[\![\Diamond]\!] = \{t \mid \ t \text{ has some } * \text{ leaf}\}$ is because if $\sqcup_i t_i$ has a $*$ leaf, then that leaf must already be contained in t_i for some i. Similarly, if $\sqcup_i t_i \in [\![\Box]\!]$ then, because $[\![\Box]\!] = \{t \,|\text{t has finite height and every leaf is a}*\}$, the tree $\sqcup_i t_i$ has finitely many leaves and all must be contained in t_i for some i. Hence $t_i \in [\![\Box]\!]$. Strong decomposability holds because:

$$\mu r \in [\![\Diamond]\!] \;\Leftrightarrow\; r[\in [\![\Diamond]\!]] \in [\![\Diamond]\!] \quad \text{and} \quad \mu r \in [\![\Box]\!] \;\Leftrightarrow\; r[\in [\![\Box]\!]] \in [\![\Box]\!].$$

The right-hand-side of the former states that r has as a leaf a tree t, which itself has a leaf $*$. That of the latter states that r is finite and all leaves are finite trees t that have only $*$ leaves. The same arguments show that $\{\Diamond\}$ and $\{\Box\}$ are also decomposable sets of Scott open modalities.

Example 3 (Probabilistic choice). $\mathcal{O} = \{\mathsf{P}_{>q} \mid q \in \mathbb{Q}, 0 \le q < 1\}$. For the Scott-openness of $[\![\mathsf{P}_{>q}]\!] = \{t \mid \mathbf{P}(\, t \text{ terminates with a * leaf}\,) \;>\; q\}$, note that $\mathbf{P}(\, \sqcup_i t_i \text{ terminates with a} * \text{leaf}\,)$ is determined by some countable sum over the leaves of t_i. If this sum is greater than a rational q, then some finite approximation of the sum must already be above q. The finite sum is over finitely many leaves from $\sqcup_i t_i$, all of which will be present in t_i for some i. Hence $t_i \in [\![\mathsf{P}_{>q}]\!]$.

We have strong decomposability, since $\mathbf{P}(\, \mu r \text{ terminates with a} * \text{leaf}\,)$ equals the integral of the function $f_r(x) = \mathit{sup}\{y \in [0,1] \,|\, r[[\![\mathsf{P}_{>x}]\!]] \in [\![\mathsf{P}_{>y}]\!]\}$ from $[0,1]$ to $[0,1]$. Indeed, $f_r(x)$ gives the probability that r return a tree $t \in [\![\mathsf{P}_{>x}]\!]$. So we know that if $\forall x, y, r[[\![\mathsf{P}_{>x}]\!]] \in [\![\mathsf{P}_{>y}]\!] \Rightarrow r'[[\![\mathsf{P}_{>x}]\!]] \in [\![\mathsf{P}_{>y}]\!]$, then $f_{r'}(x) \ge f_r(x)$ for any x. Hence if $\mu r \in [\![\mathsf{P}_{>q}]\!]$ then $\int f_r > q$, whence also $\int f_{r'} > q$, which means $\mu r' \in [\![\mathsf{P}_{>q}]\!]$.

Example 4 (Global store). We have $\mathcal{O} = \{(s \rightarrowtail s') \mid s, s' \in \mathit{State}\}$. For the Scott-openness of $[\![(s \rightarrowtail s')]\!] = \{t \mid \mathit{exec}(t,s) = (*, r)\}$, note that if $\mathit{exec}(\sqcup_i t_i, s) = (*, s')$, there is a single finite branch of t that follows the path the recursive function exec took. This branch must already be contained in t_i for some i. We also have strong decomposability since:

$$\mu r \in [\![s \rightarrowtail s']\!] \quad \Leftrightarrow \quad \exists s'' \in \mathit{State},\, r[\in [\![s'' \rightarrowtail s']\!]] \in [\![s \rightarrowtail s'']\!].$$

Which just means that $\mathit{exec}(r,s) = (t, s'')$ and $\mathit{exec}(t, s'') = (*, s')$ for some s''.

Example 5 (Input/output). We have $\mathcal{O} = \{\langle w\rangle{\downarrow}, \langle w\rangle_{\dots} \mid w$ an i/o-trace$\}$. For the Scott-openness of $[\![\langle w\rangle{\downarrow}]\!] = \{t \mid t \models \langle w\rangle{\downarrow}\{*\}\}$, note that the i/o-trace $\langle w\rangle{\downarrow}$ is given by some finite branch, which if in $\sqcup_i t_i$ must be in t_i for some i. The Scott-openness of $[\![\langle w\rangle_{\dots}]\!] = \{t \mid t \models \langle w\rangle_{\dots}\}$ holds for similar reasons. We have strong decomposability because of the implications:

$$\mu r \in [\![\langle w\rangle{\downarrow}]\!] \quad \Leftrightarrow \quad \exists v, u \text{ i/o-traces}, vu = w \wedge r[\in [\![\langle u\rangle{\downarrow}]\!]] \in [\![\langle v\rangle{\downarrow}]\!].$$

Which means r follows trace v returning t, and t follows trace u returning $*$.

$$\mu r \in [\![\langle w\rangle \dots]\!] \;\Leftrightarrow\; r[\in [\![\downarrow]\!]] \in [\![\langle w\rangle \dots]\!] \vee \exists v, u, vu = w \wedge r[\in [\![\langle u\rangle \dots]\!]] \in [\![\langle v\rangle{\downarrow}]\!].$$

Which means either r follows trace w immediately, or it follows v returning a tree that follows u.

5 Applicative $\mathcal{O}$-(bi)similarity

In this section we look at an alternative description of our logical pre-order. Central to such a definition lies the concept of a *relator* [12,25], which we use to lift a relation on value terms to a relation on computation terms. With our family of modalities $\mathcal{O}$ we can define a relator which takes a relation $\mathcal{R} \subseteq X \times Y$ and returns the relation $\mathcal{O}(\mathcal{R}) \subseteq TX \times TY$, defined by:

$$t\,\mathcal{O}(\mathcal{R})\,t' \quad \Leftrightarrow \quad \forall A \subseteq X, \forall o \in \mathcal{O},\, t[\in A] \in [\![o]\!] \Rightarrow t'[\in (\mathcal{R}^{\uparrow} A)] \in [\![o]\!].$$

Note that $\mathcal{O}(id_{\mathbf{1}}) = (\preceq)$. Following [9], we use this relation-lifting operation to define notions of applicative similarity and bisimilarity.

Definition 14. An *applicative $\mathcal{O}$-simulation* is given by a pair of relations $\mathcal{R}^v_\tau$ and $\mathcal{R}^c_\tau$ for each type τ, where $\mathcal{R}^v_\tau \subseteq \mathit{Val}(\tau)^2$ and $\mathcal{R}^c_\tau \subseteq \mathit{Com}(\tau)^2$, such that:

1. $V\mathcal{R}^v_{\mathbf{N}}W \;\Rightarrow\; (V = W)$
2. $M\mathcal{R}^c_\tau N \;\Rightarrow\; |M|\;\mathcal{O}(\mathcal{R}^v_\tau)\;|N|$
3. $V\mathcal{R}^v_{\rho\to\tau}W \;\Rightarrow\; \forall U \in \mathit{Val}(\rho),\; VU\;\mathcal{R}^c_\tau\;WU$

Applicative $\mathcal{O}$-similarity is the largest applicative $\mathcal{O}$-simulation, which is equal to the union of all applicative $\mathcal{O}$-simulations.

Definition 15. An *applicative $\mathcal{O}$-bisimulation* is a symmetric $\mathcal{O}$-simulation. The relation of *$\mathcal{O}$-bisimilarity* is the largest applicative $\mathcal{O}$-bisimulation.

Lemma 16. *Applicative $\mathcal{O}$-bisimilarity is identical to the relation of applicative $(\mathcal{O} \cap \mathcal{O}^{\mathsf{op}})$-similarity, where $t(\mathcal{O} \cap \mathcal{O}^{\mathsf{op}})(\mathcal{R})r \Leftrightarrow t\mathcal{O}(\mathcal{R})r \wedge r\mathcal{O}(\mathcal{R}^{\mathsf{op}})t$.*

Proof. Let $\mathcal{R}$ be the $\mathcal{O}$-bisimilarity, then by symmetry we have $\mathcal{R}^{\mathsf{op}} = \mathcal{R}$. So if $M\mathcal{R}N$ we have $N\mathcal{R}M$, and by the simulation rules we derive $|M|\mathcal{O}(\mathcal{R})|N|$ and $|N|\mathcal{O}(\mathcal{R})|M|$ which is what we needed.

Let $\mathcal{R}$ be the $\mathcal{O} \cap \mathcal{O}^{\mathsf{op}}$-similarity. If $M\mathcal{R}^{\mathsf{op}}N$ then $|N|(\mathcal{O} \cap \mathcal{O}^{\mathsf{op}})(\mathcal{R})|M|$ so $|N|\mathcal{O}(\mathcal{R})|M| \wedge |M|\mathcal{O}(\mathcal{R}^{\mathsf{op}})|N|$ which results in $|M|(\mathcal{O} \cap \mathcal{O}^{\mathsf{op}})(\mathcal{R}^{\mathsf{op}})|N|$. Verifying the other simulation conditions as well, we can conclude that the symmetric closure $\mathcal{R} \cup \mathcal{R}^{\mathsf{op}}$ is also a $\mathcal{O} \cap \mathcal{O}^{\mathsf{op}}$-simulation. So $\mathcal{R}$ must, as the largest such simulation, be symmetric. Hence $\mathcal{R}$ is a symmetric $\mathcal{O}$-simulation as well.

For brevity, we will leave out the word "applicative" from here on, and write o to mean its denotation $[\![o]\!]$. We also introduce brackets, writing $o[\phi]$ for $o\,\phi$. The key result now is that the maximal relation, the $\mathcal{O}$-similarity is in most cases the same object as our logical preorder. We first give a short Lemma.

Lemma 17. *For any fragment $\mathcal{L}$ of $\mathcal{V}$ closed under countable conjunction, it holds that for each value V there is a formula $\chi_V \in \mathcal{L}$ s.t. $W \models_{\mathcal{L}} \chi_V \Leftrightarrow V \sqsubseteq_{\mathcal{L}} W$.*

Proof. For each U such that $(V \not\sqsubseteq_{\mathcal{L}} U)$, choose a formula $\phi^U \in \mathcal{L}$ such that $V \models_{\mathcal{L}} \phi^U$ and $(U \not\models \phi^U)$. Then if we define $\chi_V := \bigwedge_{\{U \mid V \not\sqsubseteq_{\mathcal{L}} U\}} \phi^U$ it holds that $V \not\sqsubseteq_{\mathcal{L}} U \Leftrightarrow U \not\models \chi_V$, which is what we want.

Theorem 2 (a). *For any family of upwards closed modalities $\mathcal{O}$, we have that the logical preorder $\sqsubseteq_{\mathcal{V}^+}$ is identical to $\mathcal{O}$-similarity.*

Proof. We write $\sqsubseteq$ instead of $\sqsubseteq_{\mathcal{V}^+}$ to make room for other annotations. We first prove that our logical preorder $\sqsubseteq$ is an $\mathcal{O}$-simulation by induction on types.

1. Values of $\mathbf{N}$. If $\overline{n} \sqsubseteq^v_{\mathbf{N}} \overline{m}$, then since $\overline{n} \models \{n\}$ we have that $\overline{m} \models \{n\}$, hence $m = n$.
2. Computations of τ. Assume $M \sqsubseteq^c_\tau N$, we prove that $|M|\mathcal{O}(\sqsubseteq^v_\tau)|N|$. Take $A \subseteq \mathit{Val}(\tau)$ and $o \in \mathcal{O}$ such that $|M|[\in A] \in o$. Taking the following formula $\phi := \bigvee_{a \in A} \chi_a$ (where χ_a as in Lemma 17), then $b \models \phi \Leftrightarrow \exists a \in A, a \sqsubseteq^v_\tau b$ and $a \in A \Rightarrow a \models \phi$. So $|M|[\models \phi] \geq |M|[\in A]$, hence since o is upwards closed, $|M|[\models \phi] \in o$. By $M \sqsubseteq^c_\tau N$ we have $|N|[\in \{b \in \mathit{Val}(\tau) \mid \exists a \in A, a \sqsubseteq^v_\tau b\}] = |N|[\models \phi] \in o$. Hence we can conclude that $|M|\mathcal{O}(\sqsubseteq^v_\tau)|N|$.
3. Function values of $\rho \to \tau$, this follows from Lemma 8 and the Induction Hypothesis.

We can conclude that $\sqsubseteq$ is an $\mathcal{O}$-simulation. Now take an arbitrary $\mathcal{O}$-simulation $\mathcal{R}$. We prove by induction on types that $\mathcal{R} \subseteq (\sqsubseteq)$.

1. Values of $\mathbf{N}$. If $V\mathcal{R}^v_{\mathbf{N}}W$ then $V = W$, hence by reflexivity we get $V \sqsubseteq^v_{\mathbf{N}} W$.
2. Computations of τ. Assume $M\mathcal{R}^c_\tau N$, we prove that $M \sqsubseteq^c_\tau N$ using the characterisation from Lemma 9. Say for $o \in \mathcal{O}$ and $\phi \in \mathit{VF}(\tau)$ we have $M \models o[\phi]$. Let $A_\phi := \{a \in \mathit{Val}(\tau) \mid a \models \phi\} \subseteq \mathit{Val}(\tau)$, then $|M|[\in A_\phi] = |M|[\models \phi] \in o$ hence by $M\mathcal{R}^c_\tau N$ we derive $|N|[\in \{b \in \mathit{Val}(\tau) \mid \exists a \in A_\phi, a\mathcal{R}^v_\tau b\}] \in o$. By Induction Hypothesis on values of τ, we know that $\mathcal{R}^v_\tau \subseteq (\sqsubseteq^v_\tau)$, hence '$\exists a \in A_\phi, a\mathcal{R}^v_\tau b$' implies $b \models \phi$. We get that $|N|[\models \phi] \geq |N|[\in \{b \in \mathit{Val}(\tau) \mid \exists a \in A_\phi, a\mathcal{R}^v_\tau b\}]$, so by upwards closure of o we have $|N|[\models \phi] \in o$ meaning $N \models o[\phi]$. We conclude that $M \sqsubseteq^c_\tau N$.
3. Function values of $\rho \to \tau$, assume $V\mathcal{R}^v_{\rho\to\tau}W$. We prove $V \sqsubseteq^v_{\rho\to\tau} W$ using the characterisation from Lemma 8. Assume $V \models (U \mapsto \Phi)$ where $U \in \mathit{Val}(\rho)$ and $\Phi \in \mathit{CF}(\tau)$, so $VU \models \Phi$. By $V\mathcal{R}^v_{\rho\to\tau}W$ we have $VU\ \mathcal{R}^c_\tau\ WU$ and by Induction Hypothesis we have $\mathcal{R}^c_\tau \subseteq (\sqsubseteq^c_\tau)$, so $VU \sqsubseteq^c_\tau WU$. Hence $WU \models \Phi$ meaning $W \models (U \mapsto \Phi)$. We can conclude that $V \sqsubseteq^v_{\rho\to\tau} W$.

4. Values of $\mathbf{1}$. If $V\mathcal{R}^v_{\mathbf{1}}W$ then $V = * = W$ hence $V \sqsubseteq^v_{\mathbf{1}} W$.

In conclusion: any $\mathcal{O}$-simulation $\mathcal{R}$ is a subset of the $\mathcal{O}$-simulation $\sqsubseteq_{\mathcal{V}^+}$. So $\sqsubseteq_{\mathcal{V}^+}$ is $\mathcal{O}$-similarity. □

Alternatively, we can look at the variation of our logic with negation. This is related to applicative bisimulations.

Theorem 2 (b). *For any family of upwards closed modalities $\mathcal{O}$, we have that the logical equivalence $\equiv_{\mathcal{V}}$ is identical to $\mathcal{O}$-bisimilarity.*

Proof. Note first that $\equiv_{\mathcal{V}}$ is symmetric.

Secondly, note that since $\equiv_{\mathcal{V}}=\sqsubseteq_{\mathcal{V}}$ we know by Lemma 17, that for any V, there is a formula χ_V such that $W \models \chi_V \Leftrightarrow V \equiv_{\mathcal{V}} W$.

Using these special formulas χ_V, the rest of the proof is very similar to the proof in Theorem 2(a). Here follow the non-trivial parts of the proof, different from the previous lemma. For proving $\equiv_{\mathcal{V}}$ is an $\mathcal{O}$-simulation:

1. Computations of τ. Assume $M \equiv^c_\tau N$ and $|M|[\in A] \in o \in \mathcal{O}$. Then $M \models o[\bigvee_{V\in A}\chi_V]$ hence $N \models o[\bigvee_{V\in A}\chi_V]$ meaning $|N|[\in \{W \mid \exists V \in A, V \equiv^c_\tau W\}]$. So $|M|\mathcal{O}(\equiv^v_\tau)|N|$.
2. Functions of $\rho \to \tau$, if $V \equiv^v_{\rho\to\tau} W$ and $U \in \mathit{Val}(\rho)$. If $VU \models \Phi$, then $V \models U \mapsto \Phi$ hence $W \models U \mapsto \Phi$ so $WU \models \Phi$. Same vice versa, so $VU \equiv^c_\tau WU$.

So $\equiv_{\mathcal{V}}$ is an $\mathcal{O}$-bisimulation. Now take any $\mathcal{O}$-bisimulation $\mathcal{R}$.

1. Computations of τ, if $M\mathcal{R}N$ and $M \models o[\phi]$ then $|M|[\models \phi] \in o$ hence $|N|[\in \{W \mid \exists V \models \phi, V\mathcal{R}^v_\tau W\}] \in o$. By Induction Hypothesis, $(\mathcal{R}^v_\tau) \subseteq (\equiv^v_\tau)$ so $\{W \mid \exists V \models \phi, V\mathcal{R}^v_\tau W\} \subseteq \{W \mid \exists V \models \phi, V \equiv^v_\tau W\}$. So by upwards closure of o we get that $|N|[\in \{W \mid \exists V \models \phi, V \equiv^v_\tau W\}] \in o$ and further that $N \models o[\phi]$. We can conclude $M \equiv_{\mathcal{V}} N$.
2. Values of $\rho \to \tau$, if $V\mathcal{R}W$ and $V \models U \mapsto \Phi$, then $VU \models \Phi$ and $VU\ \mathcal{R}\ WU$ hence by Induction Hypothesis, $VU \equiv WU$ meaning $WU \models \Phi$ so $W \models U \mapsto \Phi$. If $V \models \neg(U \mapsto \Phi)$ then $\neg(VU \models \Phi)$ hence by $VU \equiv WU$ we have $\neg(WU \models \Phi)$ so $W \models \neg(U \mapsto \Phi)$. For the $\bigvee$ and $\bigwedge$ constructors, a simple Induction Step would suffice, and for higher level negation note that $\neg\bigvee\phi \Leftrightarrow \bigwedge\neg\phi$ and $\neg\bigwedge\phi \Leftrightarrow \bigvee\neg\phi$.

We can conclude that $(\mathcal{R}) \subseteq (\equiv_{\mathcal{V}})$, so $\equiv_{\mathcal{V}}$ is indeed $\mathcal{O}$-bisimilarity. □

We end this section by stating the abstract properties of our relational lifting $\mathcal{O}(\mathcal{R})$ required for the proof by Howe's method in Sect. 6 to go through. The necessary properties were identified in [9]. The contribution of this paper is that all the required properties follow from our modality-based definition of $\mathcal{O}(\mathcal{R})$. The first set of properties tell us that $\mathcal{O}(-)$ is a relator in the sense of [12]:

Lemma 18. *If the modalities from $\mathcal{O}$ are upwards closed, then $\mathcal{O}(-)$ is a relator, meaning that:*

1. *If $\mathcal{R} \subseteq X \times X$ is reflexive, then so is $\mathcal{O}(\mathcal{R})$.*
2. $\forall \mathcal{R}, \forall \mathcal{S}, \quad \mathcal{O}(\mathcal{R})\mathcal{O}(\mathcal{S}) \subseteq \mathcal{O}(\mathcal{R}\mathcal{S})$*, where $\mathcal{R}\mathcal{S}$ is relation composition.*
3. $\forall \mathcal{R}, \forall \mathcal{S}, \quad \mathcal{R} \subseteq \mathcal{S} \Rightarrow \mathcal{O}(\mathcal{R}) \subseteq \mathcal{O}(\mathcal{S})$.
4. $\forall f : X \to Z, g : Y \to W, \mathcal{R} \subseteq Z \times W, \mathcal{O}((f \times g)^{-1}\mathcal{R}) = (Tf \times Tg)^{-1}\mathcal{O}(\mathcal{R})$ *where* $(f \times g)^{-1}(\mathcal{R}) = \{(x, y) \in X \times Y \mid f(x)\mathcal{R}g(y)\}$.

The next property together with the previous lemma establishes that $\mathcal{O}(-)$ is a *monotone relator* in the sense of [25].

Lemma 19. *If the modalities from $\mathcal{O}$ are upwards closed, then $\mathcal{O}(-)$ is* monotone*, meaning for any $f : X \to Z$, $g : Y \to W$, $\mathcal{R} \subseteq X \times Y$ and $\mathcal{S} \subseteq Z \times W$:*

$$(\forall x, y, x\mathcal{R}y \Rightarrow f(x)\,\mathcal{S}\,g(y)) \wedge t\mathcal{O}(\mathcal{R})r \Rightarrow t[x \mapsto f(x)]\,\mathcal{O}(\mathcal{S})\,r[y \mapsto g(y)]$$

The relator also interacts well with the monad structure on T.

Lemma 20. *If $\mathcal{O}$ is a decomposable set of upwards closed modalities, then:*

1. $x\mathcal{R}y \;\Rightarrow\; \eta(x)\mathcal{O}(\mathcal{R})\eta(y)$;
2. $t\mathcal{O}(\mathcal{O}(\mathcal{R}))r \;\Rightarrow\; \mu t\mathcal{O}(\mathcal{R})\mu r$.

Finally, the following properties show that relator behaves well with respect to the order on trees.

Lemma 21. *If $\mathcal{O}$ only contains Scott open modalities, then:*

1. *If $\mathcal{R}$ is reflexive, then $t \leq r \Rightarrow t\mathcal{O}(\mathcal{R})r$.*
2. *For any two sequences $u_0 \leq u_1 \leq u_2 \leq \ldots$ and $v_0 \leq v_1 \leq v_2 \leq \ldots$:* $\forall n, (u_n\mathcal{O}(\mathcal{R})v_n) \;\Rightarrow\; (\sqcup_n u_n)\mathcal{O}(\mathcal{R})(\sqcup_n v_n)$

The lemmas above list the core properties of the relator, which are satisfied when our family $\mathcal{O}$ is decomposable and contains only Scott open modalities. The results below follow from those above.

Corollary 22. *If $\mathcal{O}$ contains only upwards closed modalities, then:*

$$\mathcal{O} \text{ is decomposable} \quad \Leftrightarrow \quad \forall \mathcal{R} \subseteq X \times Y, \forall t, r \in TT\mathbf{1}, (t\mathcal{O}(\mathcal{O}(\mathcal{R}))r \Rightarrow \mu t\,\mathcal{O}(\mathcal{R})\,\mu r)$$

Corollary 23. *If $\mathcal{O}$ is a decomposable family of upwards closed modalities, then lifted relations are preserved by Kleisli lifting and effect operators:*

1. *Given $f : X \to Z$, $g : Y \to W$, $\mathcal{R} \subseteq X \times Y$ and $\mathcal{S} \subseteq Z \times W$, if for all $x \in X$ and $y \in Y$ we have $x\mathcal{R}y \Rightarrow f(x)\,\mathcal{O}(\mathcal{S})\,g(y))$ and if $t\mathcal{O}(\mathcal{R})r$ then $\mu(t[x \mapsto f(x)])\,\mathcal{O}(\mathcal{S})\,\mu(r[y \mapsto g(y)])$*
2. $(\forall k, u_k\mathcal{O}(\mathcal{S})v_k) \;\Rightarrow\; \sigma(u_0, u_1, \ldots)\mathcal{O}(\mathcal{S})\sigma(v_0, v_1, \ldots)$

Point 2 of Corollary 23 has been stated in such a way that it contains both the infinite arity case $\alpha^{\mathbf{N}} \to \alpha$ and the finite arity case $\alpha^n \to \alpha$. So it states that any lifted relation is preserved under any of the predefined algebraic effects.

6 Howe's Method

In this section, we apply Howe's method, first developed in [5,6], to establish the compatibility of applicative (bi)similarity, and hence of the behavioural preorders. Given a relation $\mathcal{R}$ on terms, one defines its *Howe closure* $\mathcal{R}^\bullet$, which is compatible and contains the open extension $\mathcal{R}^\circ$. Our proof makes fundamental use of the relator properties from Sect. 5, closely following the approach of [9].

Proposition 24. *If $\mathcal{O}$ is a decomposable set of Scott open modalities, then for any $\mathcal{O}$-simulation preorder $\sqsubseteq$, the restriction of its Howe closure $\sqsubseteq^\bullet$ to closed terms is an $\mathcal{O}$-simulation.*

In the proof of the proposition, the relator properties are mainly used to show that $\sqsubseteq^\bullet$ satisfies condition (2) in Definition 14.

We can now establish the compatibility of applicative $\mathcal{O}$-similarity.

Theorem 3 (a). *If $\mathcal{O}$ is a decomposable set of Scott open modalities, then the open extension of the relation of $\mathcal{O}$-similarity is compatible.*

Proof (sketch). We write $\sqsubseteq_s$ for the relation of $\mathcal{O}$-similarity. Since $\sqsubseteq_s$ is an $\mathcal{O}$-simulation, we know by Proposition 24 that $\sqsubseteq_s^\bullet$ limited to closed terms is one as well, and hence is contained in the largest $\mathcal{O}$-simulation $\sqsubseteq_s$. Since $\sqsubseteq_s^\bullet$ is compatible, it is contained in the open extension $\sqsubseteq_s^\circ$. We can conclude that $\sqsubseteq_s^\circ$ is equal to the Howe closure $\sqsubseteq_s^\bullet$, which is compatible. $\square$

To prove that $\mathcal{O}$-bisimilarity is compatible, we use the following result from [10] (where we write $\mathcal{S}^*$ for the transitive-reflexive closure of a relation $\mathcal{S}$).

Lemma 25. *If $\mathcal{R}^\circ$ is symmetric and reflexive, then $\mathcal{R}^{\bullet *}$ is symmetric.*

Theorem 3 (b). *If $\mathcal{O}$ is a decomposable set of Scott open modalities, then the open extension of the relation of $\mathcal{O}$-bisimilarity is compatible.*

Proof (sketch). We write $\mathcal{O}$-bisimilarity as $\sqsubseteq_b$. From Proposition 24 we know that $\sqsubseteq_b^\bullet$ on closed terms is an $\mathcal{O}$-simulation, and so we know $\sqsubseteq_b^{\bullet *}$ is an $\mathcal{O}$-simulation as well (using Lemma 18). Since $\sqsubseteq_b$ is reflexive and symmetric, we know by the previous lemma that $\sqsubseteq_b^{\bullet *}$ is symmetric. Hence $\sqsubseteq_b^{\bullet *}$ is an $\mathcal{O}$-bisimulation, implying $(\sqsubseteq_b^{\bullet *}) \subseteq (\sqsubseteq_b^\circ)$ by compatibility of $\sqsubseteq_b^{\bullet *}$. Since $(\sqsubseteq_b^\circ) \subseteq (\sqsubseteq_b^\bullet) \subseteq (\sqsubseteq_b^{\bullet *})$ we have that $(\sqsubseteq_b^{\bullet *}) = (\sqsubseteq_b^\circ)$, and we can conclude that $\sqsubseteq_b^\circ$ is compatible. $\square$

Theorem 1 is an immediate consequence of Theorems 2 and 3.

7 Pure Behavioural Logic

In this section, we briefly explore an alternative formulation of our logic. This has both conceptual and practical motivations. Our very approach to behavioural logic, fits into the category of *endogenous* logics in the sense of Pnueli [24]. Formulas (ϕ and Φ) express properties of individual programs, through satisfaction relations ($V \models \phi$ and $M \models \Phi$). Programs are thus considered as 'models' of the logic, with the satisfaction relation being defined via program behaviour.

It is conceptually appealing to push the separation between program and logic to its natural conclusion, and ask for the syntax of the logic to be independent of the syntax of the programming language. Indeed, it seems natural that it should be possible to express properties of program behaviour without knowledge of the syntax of the programming language. Under our formulation of the logic $\mathcal{V}$, this desideratum is violated by the value formula $(V \mapsto \Psi)$ at function type, which mentions the programming language value V.

This issue can be addressed, by replacing the basic value formula $(V \mapsto \Psi)$ with the alternative $(\phi \mapsto \Psi)$, already mentioned in Sect. 3. Such a change also has a practical motivation. The formula $(\phi \mapsto \Psi)$ declares a precondition and postcondition for function application, supporting a useful specification style.

Definition 26. The *pure behavioural logic* $\mathcal{F}$ is defined by replacing rule (2) in Fig. 2 with the alternative:

$$\frac{\phi \in \mathit{VF}(\rho) \qquad \Psi \in \mathit{CF}(\tau)}{(\phi \mapsto \Psi) \in \mathit{VF}(\rho \to \tau)}\,(2^*)$$

The semantics is modified by defining $V \models (\phi \mapsto \Psi)$ using formula (2) of Sect. 3.

Proposition 27. *If the open extension of $\equiv_{\mathcal{V}}$ is compatible then the logics $\mathcal{V}$ and $\mathcal{F}$ are equi-expressive. Similarly, if the open extension of $\sqsubseteq_{\mathcal{V}^+}$ is compatible then the positive fragments $\mathcal{V}^+$ and $\mathcal{F}^+$ are equi-expressive.*

Proof. The definition of $(\phi \mapsto \Psi)$ within $\mathcal{V}$, given in (1) of Sect. 3, can be used as the basis of an inductive translation from $\mathcal{F}$ to $\mathcal{V}$ (and from $\mathcal{F}^+$ to $\mathcal{V}^+$).

For the reverse translation, whose correctness proof is more interesting, we give a little more detail. Every value/computation formula, ϕ/Φ, of $\mathcal{V}$ is inductively translated to a corresponding formula $\widehat{\phi}/\widehat{\Phi}$ of $\mathcal{F}$. The interesting case is:

$$(\widehat{V \mapsto \Phi}) \;:=\; (\psi_V \mapsto \widehat{\Phi}),$$

where ψ_V is a formula such that: $V \models_{\mathcal{F}} \psi_V$; and, for any ψ, if $V \models_{\mathcal{F}} \psi$ then $\psi_V \to \psi$ (meaning that $V' \models_{\mathcal{F}} \psi_V$ implies $V' \models_{\mathcal{F}} \psi$, for all V'). Such a formula ψ_V is easily constructed as a countable conjunction (cf. Lemma 17). One then proves, by induction on types, that the $\mathcal{F}$-semantics of $\widehat{\phi}$ (resp. $\widehat{\Phi}$) coincides with the $\mathcal{V}$-semantics of ϕ (resp. Φ). In the case for $(\widehat{V \mapsto \Phi})$, the induction hypothesis is used to establish that any V' satisfying $V' \models_{\mathcal{F}} \psi_V$ enjoys the property that $V' \equiv_{\mathcal{V}} V$. It then follows from the compatibility of $\equiv_{\mathcal{V}}$ that $WV' \equiv_{\mathcal{V}} WV$, for any W of appropriate type, whence $WV' \equiv_{\mathcal{F}} WV$. The rest of the proof can easily be erected around these observations. $\square$

Combining the above proposition with Theorem 1 we obtain the following.

Corollary 28. *Suppose $\mathcal{O}$ is a decomposable family of Scott-open modalities. Then $\equiv_{\mathcal{F}}$ coincides with $\equiv_{\mathcal{V}}$, and $\sqsubseteq_{\mathcal{F}^+}$ coincides with $\sqsubseteq_{\mathcal{V}^+}$. Hence the open extensions of $\equiv_{\mathcal{F}}$ and $\sqsubseteq_{\mathcal{F}^+}$ are compatible.*

We do not know any proof of the compatibility of the $\equiv_{\mathcal{F}}$ and $\sqsubseteq_{\mathcal{F}^+}$ relations that does not go via the logic $\mathcal{V}$. In particular, the compatibility property of the **fix** operator seems difficult to establish directly for $\equiv_{\mathcal{F}}$ and $\sqsubseteq_{\mathcal{F}^+}$.

8 Discussion and Related Work

The behavioural logics considered in this paper are designed for the purpose of clarifying the notion of 'behavioural property', and for defining behavioural equivalence. As infinitary propositional logics, they are not directly suited to practical applications such as specification and verification. Nevertheless, they serve as low-level logics into which more practical finitary logics can be translated. For this, the closure of the logics under infinitary propositional logic is important. For example, there are standard translations of quantifiers and least and greatest fixed points into infinitary propositional logic. Also, in the case of global store, Hoare triples translate into logical combinations of modal formulas.

Our approach, of basing logics for effects on behavioural modalities, may potentially inform the design of practical logics for specifying and reasoning about effects. For example, Pitts' *evaluation logic* was an early logic for general computational effects [18]. In the light of the general theory of modalities in the present paper, it seems natural to replace the built-in $\Box$ and $\Diamond$ modalities of evaluation logic, with effect-specific modalities, as in Sect. 3.

The *logic for algebraic effects*, of Plotkin and Pretnar [23], axiomatises effectful behaviour by means of an equational theory over the signature of effect operations, following the algebraic approach to effects advocated by Plotkin and Power [22]. Such equational axiomatisations are typically sound with respect to more than one notion of program equivalence. The logic of [23] can thus be used to soundly reason about program equivalence, but does not in itself determine a notion of program equivalence. Instead, our logic is specifically designed as a vehicle for defining program equivalence. In doing so, our modalities can be viewed as a chosen family of 'observations' that are compatible with the effects present in the language. It is the choice of modalities that determines the equational properties that the effect operations satisfy.

The logic of [23] itself makes use of modalities, called *operation modalities*, each associated with a single effect operations in Σ. It would be natural to replace these modalities, which are syntactic in nature, with behavioural modalities of the form we consider. Similarly, our behavioural modalities appear to offer a promising basis for developing a modality-based refinement-type system for algebraic effects. In general, an important advantage we see in the use of behavioural modalities is that our notion of *strong decomposability* appears related to the availability of compositional proof principles for modal properties. This is a promising avenue for future exploration.

A rather different approach to logics for effects has been proposed by Goncharov, Mossakowski and Schröder [3,16]. They assume a semantic setting in which the programming language is rich enough to contain a *pure fragment* that itself acts as a program logic. This approach is very powerful for certain effects. For example, Hoare logic can be derived in the case of global store. However, it appears not as widely adaptable across the range of effects as our approach.

Our logics exhibit certain similarities in form with the endogenous logic developed in Abramsky's *domain theory in logical form* [2]. Our motivation and approach are, however, quite different. Whereas Abramsky shows the usefulness of an axiomatic approach to a finitary logic as a way of characterising denotational equality, the present paper shows that there is a similar utility in considering an infinitary logic from a semantic perspective (based on operational semantics) as a method of defining behavioural equivalence.

The work in this paper has been carried out for fine-grained call-by-value [13], which is equivalent to call-by-value. The definitions can, however, be adapted to work for call-by-name, and even call-by-push-value [11]. Adding type constructors such as sum and product is also straightforward. We have not checked the generalisation to arbitrary recursive types, but we do not foresee any problem.

An omission from the present paper is that we have not said anything about *contextual equivalence*, which is often taken to be the default equivalence for applicative languages. In addition to determining the logically defined preorders/equivalences, the choice of the set $\mathcal{O}$ of modalities gives rise to a natural definition of *contextual preorder*, namely the largest compatible preorder that, on computations of unit type $\mathbf{1}$, is contained in the $\preceq$ relation from Sect. 4. The compatibility of $\sqsubseteq_{\mathcal{V}^+}$ established in the present paper means that we have the expected relation inclusions $\equiv_{\mathcal{V}} \subseteq \sqsubseteq_{\mathcal{V}^+} \subseteq \sqsubseteq_{\mathrm{ctxt}}$. It is an interesting question whether the logic can be restricted to characterise contextual equivalence/preorder. A more comprehensive investigation of contextual equivalence is being undertaken, in ongoing work, by Aliame Lopez and the first author.

The crucial notion of modality, in the present paper, was adapted from the notion of *observation* in [8]. The change from a set of trees of type $\mathbf{N}$ (an observation) to a set of unit-type trees (a modality) allows value formulas to be lifted to computation formulas, analogously to *predicate lifting* in coalgebra [7], which is a key characteristic of our modalities. Properties of *Scott-openness* and *decomposability* play a similar role the present paper to the role they play in [8]. However, the notion of decomposability for modalities (Definition 11) is more subtle than the corresponding notion for observations in [8].

There are certain limitations to the theory of modalities in the present paper. For example, for the combination of probability and nondeterminism, one might naturally consider modalities $\Diamond\mathsf{P}_r$ and $\Box\mathsf{P}_r$ asserting the possibility and necessity of the termination probability exceeding r. However, the decomposability property fails. It appears that this situation can be rescued by changing to a quantitative logic, with a corresponding notion of quantitative modality. This is a topic of ongoing research.

Acknowledgements. We thank Francesco Gavazzo, Aliaume Lopez and the anonymous referees for helpful discussions and comments.

References

1. Abramsky, S.: The lazy λ-calculus. In: Research Topics in Functional Programming, pp. 65–117 (1990)
2. Abramsky, S.: Domain theory in logical form. Ann. Pure Appl. Log. **51**(1–2), 1–77 (1991)
3. Goncharov, S., Schröder, L.: A relatively complete generic Hoare logic for order-enriched effects. In: Proceedings of the 28th Annual Symposium on Logic in Computer Science (LICS 2013), pp. 273–282. IEEE (2013)
4. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM (JACM) **32**(1), 137–161 (1985)
5. Howe, D.J.: Equality in lazy computation systems. In: Proceedings of the 4th IEEE Symposium on Logic in Computer Science, pp. 198–203 (1989)
6. Howe, D.J.: Proving congruence of bisimulation in functional programming languages. Inf. Comput. **124**(2), 103–112 (1996)
7. Jacobs, B.: Introduction to Coalgebra: Towards Mathematics of States and Observation. Cambridge University Press, Cambridge (2016)
8. Johann, P., Simpson, A., Voigtländer, J.: A generic operational metatheory for algebraic effects. In: Logic in Computer Science, pp. 209–218 (2010)
9. Ugo, D.L., Gavazzo, F., Levy, P.B.: Effectful applicative bisimilarity: Monads, relators, and the Howe's method. In: Logic in Computer Science, pp. 1–12 (2017)
10. Lassen, S.B.: Relational Reasoning about Functions and Nondeterminism. Ph.D. thesis, BRICS (1998)
11. Levy, P.B.: Call-by-push-value: decomposing call-by-value and call-by-name. Higher-Order Symbol. Comput. **19**(4), 377–414 (2006)
12. Levy, P.B.: Similarity quotients as final coalgebras. In: Hofmann, M. (ed.) FoSSaCS 2011. LNCS, vol. 6604, pp. 27–41. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19805-2_3
13. Levy, P.B., Power, J., Thielecke, H.: Modelling environments in call-by-value programming languages. Inf. Comput. **185**(2), 182–210 (2003)
14. Milner, R.: A Calculus of Communicating Systems. Springer, Heidelberg (1982). https://doi.org/10.1007/3-540-10235-3
15. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991)
16. Mossakowski, T., Schröder, L., Goncharov, S.: A generic complete dynamic logic for reasoning about purity and effects. Formal Aspects Comput. **22**(3–4), 363–384 (2010)
17. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981). https://doi.org/10.1007/BFb0017309
18. Pitts, A.: Evaluation logic. In: Birtwistle, G. (ed.) 4th Higher Order Workshop. Workshops in Computing, pp. 162–189. Springer, London (1990). https://doi.org/10.1007/978-1-4471-3182-3_11
19. Pitts, A.: Parametric polymorphism and operational equivalence. Math. Struct. Comput. Sci. **10**, 321–359 (2000)
20. Plotkin, G.: LCF considered as a programming language. Theor. Comput. Sci. **5**(3), 223–255 (1977)

21. Plotkin, G., Power, J.: Adequacy for algebraic effects. In: Honsell, F., Miculan, M. (eds.) FoSSaCS 2001. LNCS, vol. 2030, pp. 1–24. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45315-6_1
22. Plotkin, G., Power, J.: Notions of computation determine monads. In: Nielsen, M., Engberg, U. (eds.) FoSSaCS 2002. LNCS, vol. 2303, pp. 342–356. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45931-6_24
23. Plotkin, G., Pretnar, M.: A logic for algebraic effects. In: Proceedings of the Logic in Computer Science, pp. 118–129 (2008)
24. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on the Foundations of Computer Science, pp. 46–57 (1977)
25. Thijs, A.M.: Simulation and fixpoint semantics. Ph.D. thesis (1996)

Explicit Effect Subtyping

Amr Hany Saleh[1(✉)], Georgios Karachalias[1], Matija Pretnar[2], and Tom Schrijvers[1]

[1] Department of Computer Science, KU Leuven, Leuven, Belgium
ah.saleh@cs.kuleuven.be

[2] Faculty of Mathematics and Physics, University of Ljubljana, Ljubljana, Slovenia

Abstract. As popularity of algebraic effects and handlers increases, so does a demand for their efficient execution. Eff, an ML-like language with native support for handlers, has a subtyping-based effect system on which an effect-aware optimizing compiler could be built. Unfortunately, in our experience, implementing optimizations for Eff is overly error-prone because its core language is implicitly-typed, making code transformations very fragile.

To remedy this, we present an explicitly-typed polymorphic core calculus for algebraic effect handlers with a subtyping-based type-and-effect system. It reifies appeals to subtyping in explicit casts with coercions that witness the subtyping proof, quickly exposing typing bugs in program transformations.

Our typing-directed elaboration comes with a constraint-based inference algorithm that turns an implicitly-typed Eff-like language into our calculus. Moreover, all coercions and effect information can be erased in a straightforward way, demonstrating that coercions have no computational content.

1 Introduction

Algebraic effect handlers [17,18] are quickly maturing from a theoretical model to a practical language feature for user-defined computational effects. Yet, in practice they still incur a significant performance overhead compared to native effects.

Our earlier efforts [22] to narrow this gap with an optimising compiler from Eff [2] to OCaml showed promising results, in some cases reaching even the performance of hand-tuned code, but were very fragile and have been postponed until a more robust solution is found. We believe the main reason behind this fragility is the complexity of subtyping in combination with the implicit typing of Eff's core language, further aggravated by the "garbage collection" of subtyping constraints (see Sect. 7).[1]

[1] For other issues stemming from the same combination see issues #11 and #16 at https://github.com/matijapretnar/eff/issues/.

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 327–354, 2018.
https://doi.org/10.1007/978-3-319-89884-1_12

For efficient compilation, one must avoid the poisoning problem [26], where unification forces a pure computation to take the less precise impure type of the context (e.g. a pure and an impure branch of a conditional both receive the same impure type). Since this rules out existing (and likely simpler) effect systems for handlers based on row-polymorphism [8,12,14], we propose a polymorphic explicitly-typed calculus based on subtyping. More specifically, our contributions are as follows:

– First, in Sect. 3 we present ImpEff, a polymorphic implicitly-typed calculus for algebraic effects and handlers with a subtyping-based type-and-effect system. ImpEff is essentially a (desugared) source language as it appears in the compiler frontend of a language like Eff.
– Next, Sect. 4 presents ExEff, the core calculus, which combines explicit System F-style polymorphism with explicit coercions for subtyping in the style of Breazu-Tannen et al. [3]. This calculus comes with a type-and-effect system, a small-step operational semantics and a proof of type-safety.
– Section 5 specifies the typing-directed elaboration of ImpEff into ExEff and presents a type inference algorithm for ImpEff that produces the elaborated ExEff term as a by-product. It also establishes that the elaboration preserves typing, and that the algorithm is sound with respect to the specification and yields principal types.
– Finally, Sect. 6 defines SkelEff, which is a variant of ExEff without effect information or coercions. SkelEff is also representative of Multicore Ocaml's support for algebraic effects and handlers [6], which is a possible compilation target of Eff. By showing that the erasure from ExEff to SkelEff preserves semantics, we establish that ExEff's coercions are computationally irrelevant and that, despite the existence of multiple proofs for the same subtyping, there is no coherence problem. To enable erasure, ExEff annotates its types with *(type) skeletons*, which capture the erased counterpart and are, to our knowledge, a novel contribution.
– Our paper comes with two software artefacts: an ongoing implementation[2] of a compiler from Eff to OCaml with ExEff at its core, and an Abella mechanisation[3] of Theorems 1, 2, 6, and 7. Remaining theorems all concern the inference algorithm, and their proofs closely follow [20].

The full version of this paper includes an appendix with omitted figures and can be found at http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW711.abs.html.

2 Overview

This section presents an informal overview of the ExEff calculus, and the main issues with elaborating to and erasing from it.

[2] https://github.com/matijapretnar/eff/tree/explicit-effect-subtyping.
[3] https://github.com/matijapretnar/proofs/tree/master/explicit-effect-subtyping.

2.1 Algebraic Effect Handlers

The main premise of algebraic effects is that impure behaviour arises from a set of *operations* such as `Get` and `Set` for mutable store, `Read` and `Print` for interactive input and output, or `Raise` for exceptions [17]. This allows generalizing exception handlers to other effects, to express backtracking, co-operative multithreading and other examples in a natural way [2,18].

Assume operations $\texttt{Tick} : \texttt{Unit} \to \texttt{Unit}$ and $\texttt{Tock} : \texttt{Unit} \to \texttt{Unit}$ that take a unit value as a parameter and yield a unit value as a result. Unlike special built-in operations, these operations have no intrinsic effectful behaviour, though we can give one through handlers. For example, the handler $\{\texttt{Tick}\,x\,k \mapsto (\texttt{Print}\ \text{“tick”}; k\ \texttt{unit}), \texttt{Tock}\,x\,k \mapsto \texttt{Print}\ \text{“tock”}\}$ replaces all calls of `Tick` by printing out “tick” and similarly for `Tock`. But there is one significant difference between the two cases. Unlike exceptions, which always abort the evaluation, operations have a continuation waiting for their result. It is this continuation that the handler captures in the variable k and potentially uses in the handling clause. In the clause for `Tick`, the continuation is resumed by passing it the expected unit value, whereas in the clause for `Tock`, the operation is discarded. Thus, if we handle a computation emitting the two operations, it will print out “tick” until a first “tock” is printed, after which the evaluation stops.

2.2 Elaborating Subtyping

Consider the computation $\texttt{do}\ x \leftarrow \texttt{Tick}\ \texttt{unit}; f\ x$ and assume that f has the function type $\texttt{Unit} \to \texttt{Unit} \ !\ \{\texttt{Tock}\}$, taking unit values to unit values and perhaps calling `Tock` operations in the process. The whole computation then has the type $\texttt{Unit} \ !\ \{\texttt{Tick}, \texttt{Tock}\}$ as it returns the unit value and may call `Tick` and `Tock`.

The above typing implicitly appeals to subtyping in several places. For instance, $\texttt{Tick}\ \texttt{unit}$ has type $\texttt{Unit} \ !\ \{\texttt{Tick}\}$ and $f\ x$ type $\texttt{Unit} \ !\ \{\texttt{Tock}\}$. Yet, because they are sequenced with `do`, the type system expects they have the same set of effects. The discrepancies are implicitly reconciled by the subtyping which admits both $\{\texttt{Tick}\} \leqslant \{\texttt{Tick}, \texttt{Tock}\}$ and $\{\texttt{Tock}\} \leqslant \{\texttt{Tick}, \texttt{Tock}\}$.

We elaborate the ImpEff term into the explicitly-typed core language ExEff to make those appeals to subtyping explicit by means of casts with coercions:

$$\texttt{do}\ x \leftarrow ((\texttt{Tick}\ \texttt{unit}) \rhd \gamma_1); (f\ x) \rhd \gamma_2$$

A coercion γ is a witness for a subtyping $A \ !\ \Delta \leqslant A' \ !\ \Delta'$ and can be used to cast a term c of type $A \ !\ \Delta$ to a term $c \rhd \gamma$ of type $A' \ !\ \Delta'$. In the above term, γ_1 and γ_2 respectively witness $\texttt{Unit} \ !\ \{\texttt{Tick}\} \leqslant \texttt{Unit} \ !\ \{\texttt{Tick}, \texttt{Tock}\}$ and $\texttt{Unit} \ !\ \{\texttt{Tock}\} \leqslant \texttt{Unit} \ !\ \{\texttt{Tick}, \texttt{Tock}\}$.

2.3 Polymorphic Subtyping for Types and Effects

The above basic example only features monomorphic types and effects. Yet, our calculus also supports polymorphism, which makes it considerably more expressive. For instance the type of f in $\texttt{let}\ f = (\texttt{fun}\ g \mapsto g\ \texttt{unit})\ \texttt{in}\ \ldots$ is generalised to:

$$\forall\alpha,\alpha'.\forall\delta,\delta'.\alpha \leqslant \alpha' \Rightarrow \delta \leqslant \delta' \Rightarrow (\texttt{Unit} \to \alpha \mathrel{!} \delta) \to \alpha' \mathrel{!} \delta'$$

This polymorphic type scheme follows the qualified types convention [9] where the type $(\texttt{Unit} \to \alpha \mathrel{!} \delta) \to \alpha' \mathrel{!} \delta'$ is subjected to several qualifiers, in this case $\alpha \leqslant \alpha'$ and $\delta \leqslant \delta'$. The universal quantifiers on the outside bind the type variables α and α', and the effect set variables δ and δ'.

The elaboration of f into ExEff introduces explicit binders for both the quantifiers and the qualifiers, as well as the explicit casts where subtyping is used.

$$\Lambda\alpha.\Lambda\alpha'.\Lambda\delta.\Lambda\delta'.\Lambda(\omega\!:\!\alpha \leqslant \alpha').\Lambda(\omega'\!:\!\delta \leqslant \delta').\texttt{fun}\ (g\!:\!\texttt{Unit} \to \alpha\,!\,\delta) \mapsto (g\,\texttt{unit}) \rhd (\omega\,!\,\omega')$$

Here the binders for qualifiers introduce coercion variables ω between pure types and ω' between operation sets, which are then combined into a computation coercion $\omega \mathrel{!} \omega'$ and used for casting the function application $g\,\texttt{unit}$ to the expected type.

Suppose that h has type $\texttt{Unit} \to \texttt{Unit}\,!\,\{\texttt{Tick}\}$ and $f\,h$ type $\texttt{Unit}\,!\,\{\texttt{Tick},\texttt{Tock}\}$. In the ExEff calculus the corresponding instantiation of f is made explicit through type and coercion applications

$$f\,\texttt{Unit}\,\texttt{Unit}\,\{\texttt{Tick}\}\,\{\texttt{Tick},\texttt{Tock}\}\,\gamma_1\,\gamma_2\,h$$

where γ_1 needs to be a witness for $\texttt{Unit} \leqslant \texttt{Unit}$ and γ_2 for $\{\texttt{Tick}\} \leqslant \{\texttt{Tick},\texttt{Tock}\}$.

2.4 Guaranteed Erasure with Skeletons

One of our main requirements for ExEff is that its effect information and subtyping can be easily erased. The reason is twofold. Firstly, we want to show that neither plays a role in the runtime behaviour of ExEff programs. Secondly and more importantly, we want to use a conventionally typed (System F-like) functional language as a backend for the Eff compiler.

At first, erasure of both effect information and subtyping seems easy: simply drop that information from types and terms. But by dropping the effect variables and subtyping constraints from the type of f, we get $\forall\alpha,\alpha'.(\texttt{Unit} \to \alpha) \to \alpha'$ instead of the expected type $\forall\alpha.(\texttt{Unit} \to \alpha) \to \alpha$. In our naive erasure attempt we have carelessly discarded the connection between α and α'. A more appropriate approach to erasure would be to unify the types in dropped subtyping constraints. However, unifying types may reduce the number of type variables when they become instantiated, so corresponding binders need to be dropped, greatly complicating the erasure procedure and its meta-theory.

Fortunately, there is an easier way by tagging all bound type variables with *skeletons*, which are barebone types without effect information. For example, the skeleton of a function type $A \to B \mathrel{!} \Delta$ is $\tau_1 \to \tau_2$, where τ_1 is the skeleton of

A and τ_2 the skeleton of B. In ExEff every well-formed type has an associated skeleton, and any two types $A_1 \leqslant A_2$ share the same skeleton. In particular, binders for type variables are explicitly annotated with skeleton variables ς. For instance, the actual type of f is:

$$\forall\varsigma.\forall(\alpha : \varsigma), (\alpha' : \varsigma).\forall\delta, \delta'.\alpha \leqslant \alpha' \Rightarrow \delta \leqslant \delta' \Rightarrow (\mathtt{Unit} \to \alpha \mathrel{!} \delta) \to \alpha' \mathrel{!} \delta'$$

The skeleton quantifications and annotations also appear at the term-level:

$$\Lambda\varsigma.\Lambda(\alpha : \varsigma).\Lambda(\alpha' : \varsigma).\Lambda\delta.\Lambda\delta'.\Lambda(\omega : \alpha \leqslant \alpha').\Lambda(\omega' : \delta \leqslant \delta')\ldots.$$

Now erasure is really easy: we drop not only effect and subtyping-related term formers, but also type binders and application. We do retain skeleton binders and applications, which take over the role of (plain) types in the backend language. In terms, we replace types by their skeletons. For instance, for f we get:

$$\Lambda\varsigma.\mathtt{fun}\ (g : \mathtt{Unit} \to \varsigma) \mapsto g\,\mathtt{unit} \quad : \quad \forall\varsigma.(\mathtt{Unit} \to \varsigma) \to \varsigma$$

Terms

$$\begin{array}{rcl}
\text{value } v & ::= & x \mid \mathtt{unit} \mid \mathtt{fun}\ x \mapsto c \mid h \\
\text{handler } h & ::= & \{\mathtt{return}\ x \mapsto c_r, \mathtt{Op}_1\ x\, k \mapsto c_{\mathtt{Op}_1}, \ldots, \mathtt{Op}_n\ x\, k \mapsto c_{\mathtt{Op}_n}\} \\
\text{computation } c & ::= & \mathtt{return}\ v \mid \mathtt{Op}\ v\ (y.c) \mid \mathtt{do}\ x \leftarrow c_1; c_2 \\
 & \mid & \mathtt{handle}\ c\ \mathtt{with}\ v \mid v_1\ v_2 \mid \mathtt{let}\ x = v\ \mathtt{in}\ c
\end{array}$$

Types & Constraints

$$\begin{array}{rcl}
\text{skeleton } \tau & ::= & \varsigma \mid \mathtt{Unit} \mid \tau_1 \to \tau_2 \mid \tau_1 \Rrightarrow \tau_2 \\
 & & \\
\text{value type } A, B & ::= & \alpha \mid \mathtt{Unit} \mid A \to \underline{C} \mid \underline{C} \Rrightarrow \underline{D} \\
\text{qualified type } K & ::= & A \mid \pi \Rightarrow K \\
\text{polytype } S & ::= & K \mid \forall\varsigma.S \mid \forall\alpha{:}\tau.S \mid \forall\delta.S \\
\text{computation type } \underline{C}, \underline{D} & ::= & A \mathrel{!} \Delta \\
\text{dirt } \Delta & ::= & \delta \mid \emptyset \mid \{\mathtt{Op}\} \cup \Delta \\
 & & \\
\text{simple constraint } \pi & ::= & A_1 \leqslant A_2 \mid \Delta_1 \leqslant \Delta_2 \\
\text{constraint } \rho & ::= & \pi \mid \underline{C} \leqslant \underline{D}
\end{array}$$

Fig. 1. ImpEff Syntax

3 The ImpEff Language

This section presents ImpEff, a basic functional calculus with support for algebraic effect handlers, which forms the core language of our optimising compiler. We describe the relevant concepts, but refer the reader to Pretnar's tutorial [21], which explains essentially the same calculus in more detail.

3.1 Syntax

Figure 1 presents the syntax of the source language. There are two main kinds of terms: (pure) values v and (dirty) computations c, which may call effectful operations. Handlers h are a subsidiary sort of values. We assume a given set of *operations* $\texttt{Op}$, such as $\texttt{Get}$ and $\texttt{Put}$. We abbreviate $\texttt{Op}_1\, x\, k \mapsto c_{\texttt{Op}_1}, \ldots, \texttt{Op}_n\, x\, k \mapsto c_{\texttt{Op}_n}$ as $[\texttt{Op}\, x\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}$, and write $\mathcal{O}$ to denote the set $\{\texttt{Op}_1, \ldots, \texttt{Op}_n\}$.

Similarly, we distinguish between two basic sorts of types: the value types A, B and the computation types $\underline{C}, \underline{D}$. There are four forms of value types: type variables α, function types $A \to \underline{C}$, handler types $\underline{C} \Rrightarrow \underline{D}$ and the $\texttt{Unit}$ type. Skeletons τ capture the shape of types, so, by design, their forms are identical. The computation type $A \mathbin{!} \Delta$ is assigned to a computation returning values of type A and potentially calling operations from the *dirt* set Δ. A dirt set contains zero or more operations $\texttt{Op}$ and is terminated either by an empty set or a dirt variable δ. Though we use $\texttt{cons}$-list syntax, the intended semantics of dirt sets Δ is that the order of operations $\texttt{Op}$ is irrelevant. Similarly to all HM-based systems, we discriminate between value types (or monotypes) A, qualified types K and polytypes (or type schemes) S. (Simple) subtyping constraints π denote inequalities between either value types or dirts. We also present the more general form of constraints ρ that includes inequalities between computation types (as we illustrate in Sect. 3.2 below, this allows for a single, uniform constraint entailment relation). Finally, polytypes consist of zero or more skeleton, type or dirt abstractions followed by a qualified type.

3.2 Typing

Figure 2 presents the typing rules for values and computations, along with a typing-directed elaboration into our target language ExEff. In order to simplify the presentation, in this section we focus exclusively on typing. The parts of the rules that concern elaboration are highlighted in gray and are discussed in Sect. 5.

Values. Typing for values takes the form $\Gamma \vdash_v v : A \rightsquigarrow v'$, and, given a typing environment Γ, checks a value v against a value type A.

Rule TmVar handles term variables. Given that x has type $(\forall \overline{\varsigma}. \overline{\alpha : \tau}. \forall \overline{\delta}. \overline{\pi} \Rightarrow A)$, we *appropriately* instantiate the skeleton $(\overline{\varsigma})$, type $(\overline{\alpha})$, and dirt $(\overline{\delta})$ variables, and ensure that the instantiated wanted constraints $\overline{\sigma(\pi)}$ are satisfied, via side condition $\overline{\Gamma \vdash_{\text{co}} \gamma : \sigma(\pi)}$. Rule TmCastV allows casting the type of a value v from A to B, if A is a subtype of B (upcasting). As illustrated by Rule TmTmAbs, we omit freshness conditions by adopting the Barendregt convention [1]. Finally, Rule TmHand gives typing for handlers. It requires that the right-hand sides of the return clause and all operation clauses have the same computation type $(B \mathbin{!} \Delta)$, and that all operations mentioned are part of the top-level signature Σ.[4] The result type takes the form $A \mathbin{!} \Delta \cup \mathcal{O} \Rrightarrow B \mathbin{!} \Delta$, capturing the intended handler semantics: given a computation of type $A \mathbin{!} \Delta \cup \mathcal{O}$, the handler (a) produces a result of type B, (b) handles operations $\mathcal{O}$, and (c) propagates unhandled operations Δ to the output.

[4] We capture all defined operations along with their types in a global signature Σ.

typing environment $\Gamma ::= \epsilon \mid \Gamma, \varsigma \mid \Gamma, \alpha : \tau \mid \Gamma, \delta \mid \Gamma, x : S \mid \Gamma, \omega : \pi$

$\boxed{\Gamma \vdash_v v : A \rightsquigarrow v'}$ **Values**

$$\frac{(x : \forall \overline{\varsigma}. \forall \overline{\alpha : \tau}. \forall \overline{\delta}. \overline{\pi} \Rightarrow A) \in \Gamma \qquad \sigma = [\overline{\tau' / \varsigma}, \overline{B / \alpha}, \overline{\Delta / \delta}] \qquad \overline{\Gamma \vdash_{\mathsf{co}} \gamma : \sigma(\pi)}}{\Gamma \vdash_v x : \sigma(A) \rightsquigarrow x\ \ \overline{\tau'}\ \overline{B}\ \overline{\Delta}\ \overline{\gamma}}\ \textsc{TmVar}$$

$$\frac{\begin{array}{c}\Gamma \vdash_v v : A \rightsquigarrow v' \\ \Gamma \vdash_{\mathsf{co}} \gamma : A \leqslant B\end{array}}{\Gamma \vdash_v v : B \rightsquigarrow v' \rhd \gamma}\ \textsc{TmCastV} \qquad \frac{}{\Gamma \vdash_v \mathtt{unit} : \mathsf{Unit} \rightsquigarrow \mathtt{unit}}\ \textsc{TmUnit}$$

$$\frac{\Gamma, x : A \vdash_c c : \underline{C} \rightsquigarrow c' \qquad \Gamma \vdash_{\mathsf{vty}} A : \tau \rightsquigarrow T}{\Gamma \vdash_v (\mathtt{fun}\ x \mapsto c) : A \to \underline{C} \rightsquigarrow \mathtt{fun}\ (x : T) \mapsto c'}\ \textsc{TmTmAbs}$$

$$\frac{\begin{array}{c}\Gamma, x : A \vdash_c c_r : B \mathbin{!} \Delta \rightsquigarrow c'_r \qquad \Gamma \vdash_{\mathsf{vty}} A : \tau \rightsquigarrow T \\ \left[(\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma \quad \Gamma, x : A_{\mathtt{Op}}, k : B_{\mathtt{Op}} \to B \mathbin{!} \Delta \vdash_c c_{\mathtt{Op}} : B \mathbin{!} \Delta \rightsquigarrow c'_{\mathtt{Op}}\right]_{\mathtt{Op} \in \mathcal{O}} \\ c_{res} = \{\mathtt{return}\ (x : T) \mapsto c'_r, [\mathtt{Op}\, x\, k \mapsto c'_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\}\end{array}}{\Gamma \vdash_v \{\mathtt{return}\ x \mapsto c_r, [\mathtt{Op}\, x\, k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : A \mathbin{!} \Delta \cup \mathcal{O} \Rrightarrow B \mathbin{!} \Delta \rightsquigarrow c_{res}}\ \textsc{TmHand}$$

$\boxed{\Gamma \vdash_c c : \underline{C} \rightsquigarrow c'}$ **Computations**

$$\frac{\begin{array}{c}\Gamma \vdash_c c : \underline{C}_1 \rightsquigarrow c' \\ \Gamma \vdash_{\mathsf{co}} \gamma : \underline{C}_1 \leqslant \underline{C}_2\end{array}}{\Gamma \vdash_c c : \underline{C}_2 \rightsquigarrow c' \rhd \gamma}\ \textsc{TmCastC} \qquad \frac{\begin{array}{c}\Gamma \vdash_v v_1 : A \to \underline{C} \rightsquigarrow v'_1 \\ \Gamma \vdash_v v_2 : A \rightsquigarrow v'_2\end{array}}{\Gamma \vdash_c v_1\ v_2 : \underline{C} \rightsquigarrow v'_1\ v'_2}\ \textsc{TmTmApp}$$

$$\frac{\begin{array}{c}S = \forall \overline{\varsigma}. \overline{\alpha : \tau}. \forall \overline{\delta}. \overline{\pi} \Rightarrow A \\ \Gamma, \overline{\varsigma}, \overline{\alpha : \tau}, \overline{\delta}, \overline{\omega : \pi} \vdash_v v : A \rightsquigarrow v' \qquad \Gamma, x : S \vdash_c c : \underline{C} \rightsquigarrow c'\end{array}}{\Gamma \vdash_c \mathtt{let}\ x = v\ \mathtt{in}\ c : \underline{C} \rightsquigarrow \mathtt{let}\ x = \Lambda \overline{\varsigma}. \Lambda \overline{\alpha : \tau}. \Lambda \overline{\delta}. \Lambda (\overline{\omega : \pi}). v'\ \mathtt{in}\ c'}\ \textsc{TmLet}$$

$$\frac{\Gamma \vdash_v v : A \rightsquigarrow v'}{\Gamma \vdash_c \mathtt{return}\ v : A \mathbin{!} \emptyset \rightsquigarrow \mathtt{return}\ v'}\ \textsc{TmReturn}$$

$$\frac{\begin{array}{c}(\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma \qquad \Gamma \vdash_v v : A_{\mathtt{Op}} \rightsquigarrow v' \\ \Gamma, y : B_{\mathtt{Op}} \vdash_c c : A \mathbin{!} \Delta \rightsquigarrow c' \qquad \Gamma \vdash_{\mathsf{vty}} B_{\mathtt{Op}} : \tau \rightsquigarrow T_{\mathtt{Op}} \qquad \mathtt{Op} \in \Delta\end{array}}{\Gamma \vdash_c \mathtt{Op}\ v\ (y.c) : A \mathbin{!} \Delta \rightsquigarrow \mathtt{Op}\ v'\ (y : T_{\mathtt{Op}}.c')}\ \textsc{TmOp}$$

$$\frac{\Gamma \vdash_c c_1 : A \mathbin{!} \Delta \rightsquigarrow c'_1 \qquad \Gamma, x : A \vdash_c c_2 : B \mathbin{!} \Delta \rightsquigarrow c'_2}{\Gamma \vdash_c \mathtt{do}\ x \leftarrow c_1 ; c_2 : B \mathbin{!} \Delta \rightsquigarrow \mathtt{do}\ x \leftarrow c'_1 ; c'_2}\ \textsc{TmDo}$$

$$\frac{\Gamma \vdash_v v : \underline{C} \Rrightarrow \underline{D} \rightsquigarrow v' \qquad \Gamma \vdash_c c : \underline{C} \rightsquigarrow c'}{\Gamma \vdash_c \mathtt{handle}\ c\ \mathtt{with}\ v : \underline{D} \rightsquigarrow \mathtt{handle}\ c'\ \mathtt{with}\ v'}\ \textsc{TmHandle}$$

Fig. 2. ImpEff Typing & Elaboration

Computations. Typing for computations takes the form $\Gamma \vdash_c c : \underline{C} \rightsquigarrow c'$, and, given a typing environment Γ, checks a computation c against a type $\underline{C}$.

Rule TmCastC behaves like Rule TmCastV, but for computation types. Rule TmLet handles polymorphic, non-recursive let-bindings. Rule TmReturn handles `return` v computations. Keyword `return` effectively lifts a value v of type A into a computation of type $A \,!\, \emptyset$. Rule TmOp checks operation calls. First, we ensure that v has the appropriate type, as specified by the signature of `Op`. Then, the continuation $(y.c)$ is checked. The side condition $\texttt{Op} \in \Delta$ ensures that the called operation `Op` is captured in the result type. Rule TmDo handles sequencing. Given that c_1 has type $A \,!\, \Delta$, the pure part of the result of type A is bound to term variable x, which is brought in scope for checking c_2. As we mentioned in Sect. 2, all computations in a `do`-construct should have the same effect set, Δ. Rule TmHandle eliminates handler types, just as Rule TmTmApp eliminates arrow types.

Constraint Entailment. The specification of constraint entailment takes the form $\Gamma \vdash_{\mathsf{co}} \gamma : \rho$ and is presented in Fig. 3. Notice that we use ρ instead of π, which allows us to capture subtyping between two value types, computation types or dirts, within the same relation. Subtyping can be established in several ways:

Rule CoVar handles given assumptions. Rules VCoRefl and DCoRefl express that subtyping is reflexive, for both value types and dirts. Notice that we do not have a rule for the reflexivity of computation types since, as we illustrate below, it can be established using the reflexivity of their subparts. Rules VCoTrans, CCoTrans and DCoTrans express the transitivity of subtyping for value types, computation types and dirts, respectively. Rule VCoArr establishes inequality of arrow types. As usual, the arrow type constructor is contravariant in the argument type. Rules VCoArrL and CCoArrR are the inversions of Rule VCoArr, allowing us to establish the relation between the subparts of the arrow types. Rules VCoHand, CCoHL, and CCoHR work similarly, for handler types. Rule CCoComp captures the covariance of type constructor (!), establishing subtyping between two computation types if subtyping is established for their respective subparts. Rules VCoPure and DCoImpure are its inversions. Finally, Rules DCoNil and DCoOp establish subtyping between dirts. Rule DCoNil captures that the empty dirty set $\emptyset$ is a subdirt of any dirt Δ and Rule DCoOp expresses that dirt subtyping preserved under extension with the same operation `Op`.

Well-Formedness of Types, Constraints, Dirts, and Skeletons. The relations $\Gamma \vdash_{\mathsf{vty}} A : \tau \rightsquigarrow T$ and $\Gamma \vdash_{\mathsf{cty}} \underline{C} : \tau \rightsquigarrow \underline{C}$ check the well-formedness of value and computation types respectively. Similarly, relations $\Gamma \vdash_{\mathsf{ct}} \rho \rightsquigarrow \rho$ and $\Gamma \vdash_{\Delta} \Delta$ check the well-formedness of constraints and dirts, respectively.

$\boxed{\Gamma \vdash_{\mathsf{co}} \gamma : \rho}$ **Constraint Entailment**

$$\frac{(\omega : \pi) \in \Gamma}{\Gamma \vdash_{\mathsf{co}} \omega : \pi}\ \textsc{CoVar} \qquad \frac{\Gamma \vdash_{\mathsf{vty}} A : \tau \rightsquigarrow T}{\Gamma \vdash_{\mathsf{co}} \langle T \rangle : A \leqslant A}\ \textsc{VCoRefl}$$

$$\frac{\Gamma \vdash_{\Delta} \Delta}{\Gamma \vdash_{\mathsf{co}} \langle \Delta \rangle : \Delta \leqslant \Delta}\ \textsc{DCoRefl} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : A_1 \leqslant A_2 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : A_2 \leqslant A_3}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \gg \gamma_2 : A_1 \leqslant A_3}\ \textsc{VCoTrans}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : \underline{C}_1 \leqslant \underline{C}_2 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \underline{C}_2 \leqslant \underline{C}_3}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \gg \gamma_2 : \underline{C}_1 \leqslant \underline{C}_3}\ \textsc{CCoTrans} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : \Delta_1 \leqslant \Delta_2 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \Delta_2 \leqslant \Delta_3}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \gg \gamma_2 : \Delta_1 \leqslant \Delta_3}\ \textsc{DCoTrans}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : B \leqslant A \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \underline{C} \leqslant \underline{D}}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \to \gamma_2 : A \to \underline{C} \leqslant B \to \underline{D}}\ \textsc{VCoArr}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : A \to \underline{C} \leqslant B \to \underline{D}}{\Gamma \vdash_{\mathsf{co}} \mathit{left}(\gamma) : B \leqslant A}\ \textsc{VCoArrL} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : A \to \underline{C} \leqslant B \to \underline{D}}{\Gamma \vdash_{\mathsf{co}} \mathit{right}(\gamma) : \underline{C} \leqslant \underline{D}}\ \textsc{CCoArrR}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : \underline{C}_2 \leqslant \underline{C}_1 \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \underline{D}_1 \leqslant \underline{D}_2}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \Rrightarrow \gamma_2 : \underline{C}_1 \Rrightarrow \underline{D}_1 \leqslant \underline{C}_2 \Rrightarrow \underline{D}_2}\ \textsc{VCoHand}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \underline{C}_1 \Rrightarrow \underline{D}_1 \leqslant \underline{C}_2 \Rrightarrow \underline{D}_2}{\Gamma \vdash_{\mathsf{co}} \mathit{left}(\gamma) : \underline{C}_2 \leqslant \underline{C}_1}\ \textsc{CCoHL} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : \underline{C}_1 \Rrightarrow \underline{D}_1 \leqslant \underline{C}_2 \Rrightarrow \underline{D}_2}{\Gamma \vdash_{\mathsf{co}} \mathit{right}(\gamma) : \underline{D}_1 \leqslant \underline{D}_2}\ \textsc{CCoHR}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : A_1 \leqslant A_2 \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \Delta_1 \leqslant \Delta_2}{\Gamma \vdash_{\mathsf{co}} \gamma_1\ !\ \gamma_2 : A_1\ !\ \Delta_1 \leqslant A_2\ !\ \Delta_2}\ \textsc{CCoComp}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : A_1\ !\ \Delta_1 \leqslant A_2\ !\ \Delta_2}{\Gamma \vdash_{\mathsf{co}} \mathit{pure}(\gamma) : A_1 \leqslant A_2}\ \textsc{VCoPure} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : A_1\ !\ \Delta_1 \leqslant A_2\ !\ \Delta_2}{\Gamma \vdash_{\mathsf{co}} \mathit{impure}(\gamma) : \Delta_1 \leqslant \Delta_2}\ \textsc{DCoImpure}$$

$$\frac{}{\Gamma \vdash_{\mathsf{co}} \emptyset_{\Delta} : \emptyset \leqslant \Delta}\ \textsc{DCoNil} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : \Delta_1 \leqslant \Delta_2 \qquad (\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma}{\Gamma \vdash_{\mathsf{co}} \{\mathtt{Op}\} \cup \gamma : \{\mathtt{Op}\} \cup \Delta_1 \leqslant \{\mathtt{Op}\} \cup \Delta_2}\ \textsc{DCoOp}$$

Fig. 3. ImpEff Constraint Entailment

4 The ExEff Language

4.1 Syntax

Figure 4 presents ExEff's syntax. ExEff is an intensional type theory akin to System F [7], where every term encodes its own typing derivation. In essence, all abstractions and applications that are implicit in ImpEff, are made explicit in ExEff via new syntactic forms. Additionally, ExEff is impredicative, which is reflected in the lack of discrimination between value types, qualified types and

Terms

$$\begin{array}{rcl}
\textsf{value } v & ::= & x \mid \texttt{unit} \mid \texttt{fun}\ (x : T) \mapsto c \mid h \\
& \mid & \Lambda\varsigma.v \mid v\ \tau \mid \Lambda\alpha : \tau.v \mid v\ \ T \mid \Lambda\delta.v \mid v\ \Delta \mid \Lambda(\omega : \pi).v \mid v\ \gamma \mid v \rhd \gamma \\
\textsf{handler } h & ::= & \{\texttt{return}\ (x : T) \mapsto c_r, \texttt{Op}_1\ x\ k \mapsto c_{\texttt{Op}_1}, \ldots, \texttt{Op}_n\ x\ k \mapsto c_{\texttt{Op}_n}\} \\
\textsf{computation } c & ::= & \texttt{return}\ v \mid \texttt{Op}\ v\ (y : T.c) \mid \texttt{do}\ x \leftarrow c_1; c_2 \\
& \mid & \texttt{handle}\ c\ \texttt{with}\ v \mid v_1\ v_2 \mid \texttt{let}\ x = v\ \texttt{in}\ c \mid c \rhd \gamma
\end{array}$$

Types

$$\begin{array}{rcl}
\textsf{skeleton } \tau & ::= & \varsigma \mid \texttt{Unit} \mid \tau_1 \to \tau_2 \mid \tau_1 \Rrightarrow \tau_2 \mid \forall\varsigma.\tau \\
\\
\textsf{value type } T & ::= & \alpha \mid \texttt{Unit} \mid T \to \underline{C} \mid \underline{C}_1 \Rrightarrow \underline{C}_2 \mid \forall\varsigma.T \mid \forall\alpha{:}\tau.T \mid \forall\delta.T \mid \pi \Rightarrow T \\
\textsf{simple coercion type } \pi & ::= & T_1 \leqslant T_2 \mid \Delta_1 \leqslant \Delta_2 \\
\textsf{coercion type } \rho & ::= & \pi \mid \underline{C}_1 \leqslant \underline{C}_2 \\
\\
\textsf{computation type } \underline{C} & ::= & T\ !\ \Delta \\
\textsf{dirt } \Delta & ::= & \delta \mid \emptyset \mid \{\texttt{Op}\} \cup \Delta
\end{array}$$

Coercions

$$\begin{array}{rl}
\gamma ::= & \omega \mid \gamma_1 \gg \gamma_2 \mid \langle T \rangle \mid \gamma_1 \to \gamma_2 \mid \gamma_1 \Rrightarrow \gamma_2 \mid \textit{left}(\gamma) \mid \textit{right}(\gamma) \mid \langle \Delta \rangle \mid \emptyset_\Delta \mid \{\texttt{Op}\} \cup \gamma \\
\mid & \forall\varsigma.\gamma \mid \gamma[\tau] \mid \forall\alpha.\gamma \mid \gamma[T] \mid \forall\delta.\gamma \mid \gamma[\Delta] \mid \pi \Rightarrow \gamma \mid \gamma_1 @ \gamma_2 \mid \gamma_1\ !\ \gamma_2 \mid \textit{pure}(\gamma) \mid \textit{impure}(\gamma)
\end{array}$$

Fig. 4. ExEff Syntax

type schemes; all non-computation types are denoted by T. While the impredicativity is not strictly required for the purpose at hand, it makes for a cleaner system.

Coercions. Of particular interest is the use of explicit *subtyping coercions*, denoted by γ. ExEff uses these to replace the implicit casts of ImpEff (Rules TmCastV and TmCastC in Fig. 2) with explicit casts $(v \rhd \gamma)$ and $(c \rhd \gamma)$.

Essentially, coercions γ are explicit witnesses of subtyping derivations: each coercion form corresponds to a subtyping rule. Subtyping forms a partial order, which is reflected in coercion forms $\gamma_1 \gg \gamma_2$, $\langle T \rangle$, and $\langle \Delta \rangle$. Coercion form $\gamma_1 \gg \gamma_2$ captures transitivity, while forms $\langle T \rangle$ and $\langle \Delta \rangle$ capture reflexivity for value types and dirts (reflexivity for computation types can be derived from these).

Subtyping for skeleton abstraction, type abstraction, dirt abstraction, and qualification is witnessed by forms $\forall\varsigma.\gamma$, $\forall\alpha.\gamma$, $\forall\delta.\gamma$, and $\pi \Rightarrow \gamma$, respectively. Similarly, forms $\gamma[\tau]$, $\gamma[T]$, $\gamma[\Delta]$, and $\gamma_1 @ \gamma_2$ witness subtyping of skeleton instantiation, type instantiation, dirt instantiation, and coercion application, respectively.

Syntactic forms $\gamma_1 \to \gamma_2$ and $\gamma_1 \Rrightarrow \gamma_2$ capture injection for the arrow and the handler type constructor, respectively. Similarly, inversion forms *left*(γ) and *right*(γ) capture projection, following from the injectivity of both type constructors.

Coercion form $\gamma_1 \,!\, \gamma_2$ witnesses subtyping for computation types, using proofs for their components. Inversely, syntactic forms $\mathit{pure}(\gamma)$ and $\mathit{impure}(\gamma)$ witness subtyping between the value- and dirt-components of a computation coercion.

Finally, coercion forms $\emptyset_\Delta$ and $\{\mathtt{Op}\} \cup \gamma$ are concerned with dirt subtyping. Form $\emptyset_\Delta$ witnesses that the empty dirt $\emptyset$ is a subdirt of any dirt Δ. Lastly, coercion form $\{\mathtt{Op}\} \cup \gamma$ witnesses that subtyping between dirts is preserved under extension with a new operation. Note that we do not have an inversion form to extract a witness for $\Delta_1 \leqslant \Delta_2$ from a coercion for $\{\mathtt{Op}\} \cup \Delta_1 \leqslant \{\mathtt{Op}\} \cup \Delta_2$. The reason is that dirt sets are sets and not inductive structures. For instance, for $\Delta_1 = \{\mathtt{Op}\}$ and $\Delta_2 = \emptyset$ the latter subtyping holds, but the former does not.

4.2 Typing

Value and Computation Typing. Typing for ExEff values and computations is presented in Figs. 5 and 6 and is given by two mutually recursive relations of the form $\Gamma \vdash_{\mathsf{v}} v : T$ (values) and $\Gamma \vdash_{\mathsf{c}} c : \underline{C}$ (computations). ExEff typing environments Γ contain bindings for variables of all sorts:

$$\Gamma ::= \epsilon \mid \Gamma, \varsigma \mid \Gamma, \alpha : \tau \mid \Gamma, \delta \mid \Gamma, x : T \mid \Gamma, \omega : \pi$$

Typing is entirely syntax-directed. Apart from the typing rules for skeleton, type, dirt, and coercion abstraction (and, subsequently, skeleton, type, dirt, and coercion application), the main difference between typing for ImpEff and ExEff lies in the explicit cast forms, $(v \rhd \gamma)$ and $(c \rhd \gamma)$. Given that a value v has type T_1 and that γ is a proof that T_1 is a subtype of T_2, we can upcast v with an explicit cast operation $(v \rhd \gamma)$. Upcasting for computations works analogously.

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash_{\mathsf{v}} x : T} \qquad \frac{}{\Gamma \vdash_{\mathsf{v}} \mathtt{unit} : \mathtt{Unit}} \qquad \frac{\Gamma, x : T \vdash_{\mathsf{c}} c : \underline{C} \qquad \Gamma \vdash_{T} T : \tau}{\Gamma \vdash_{\mathsf{v}} (\mathtt{fun}\ x : T \mapsto c) : T \to \underline{C}}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} v : T_1 \qquad \Gamma \vdash_{\mathsf{co}} \gamma : T_1 \leqslant T_2}{\Gamma \vdash_{\mathsf{v}} v \rhd \gamma : T_2} \qquad \frac{\Gamma, \varsigma \vdash_{\mathsf{v}} v : T}{\Gamma \vdash_{\mathsf{v}} \Lambda\varsigma.v : \forall\varsigma.T} \qquad \frac{\Gamma, \alpha : \tau \vdash_{\mathsf{v}} v : T}{\Gamma \vdash_{\mathsf{v}} \Lambda\alpha : \tau.v : \forall\alpha : \tau.T}$$

$$\frac{\Gamma, \delta \vdash_{\mathsf{v}} v : T}{\Gamma \vdash_{\mathsf{v}} \Lambda\delta.v : \forall\delta.T} \qquad \frac{\Gamma, \omega : \pi \vdash_{\mathsf{v}} v : T \qquad \Gamma \vdash_{\rho} \pi}{\Gamma \vdash_{\mathsf{v}} \Lambda(\omega : \pi).v : \pi \Rightarrow T} \qquad \frac{\Gamma \vdash_{\mathsf{v}} v : \pi \Rightarrow T \qquad \Gamma \vdash_{\mathsf{co}} \gamma : \pi}{\Gamma \vdash_{\mathsf{v}} v\ \gamma : T}$$

$$\frac{\begin{array}{c}\Gamma, x : T_x \vdash_{\mathsf{c}} c_r : T \,!\, \Delta \\ {[(\mathtt{Op} : T_1 \to T_2) \in \Sigma \qquad \Gamma, x : T_1, k : T_2 \to T \,!\, \Delta \vdash_{\mathsf{c}} c_{\mathtt{Op}} : T \,!\, \Delta]}_{\mathtt{Op} \in \mathcal{O}}\end{array}}{\Gamma \vdash_{\mathsf{v}} \{\mathtt{return}\ (x : T_x) \mapsto c_r, [\mathtt{Op}\, x\, k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : T_x \,!\, \Delta \cup \mathcal{O} \Rrightarrow T \,!\, \Delta}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{v}} v : \forall\varsigma.T \\ \Gamma \vdash_{\tau} \tau\end{array}}{\Gamma \vdash_{\mathsf{v}} v\ \tau : T[\tau/\varsigma]} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{v}} v : \forall\alpha : \tau.T_1 \\ \Gamma \vdash_{T} T_2 : \tau\end{array}}{\Gamma \vdash_{\mathsf{v}} v\ T_2 : T_1[T_2/\alpha]} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{v}} v : \forall\delta.T \\ \Gamma \vdash_{\Delta} \Delta\end{array}}{\Gamma \vdash_{\mathsf{v}} v\ \Delta : T[\Delta/\delta]}$$

Fig. 5. ExEff Value Typing

Well-Formedness of Types, Constraints, Dirts and Skeletons. The definitions of the judgements that check the well-formedness of ExEff value types ($\Gamma \vdash_{T} T : \tau$), computation types ($\Gamma \vdash_{\underline{C}} \underline{C} : \tau$), dirts ($\Gamma \vdash_{\Delta} \Delta$), and skeletons ($\Gamma \vdash_{\tau} \tau$) are equally straightforward as those for ImpEff.

Coercion Typing. Coercion typing formalizes the intuitive interpretation of coercions we gave in Sect. 4.1 and takes the form $\Gamma \vdash_{co} \gamma : \rho$. It is essentially an extension of the constraint entailment relation of Fig. 3.

4.3 Operational Semantics

Figure 7 presents selected rules of ExEff's small-step, call-by-value operational semantics. For lack of space, we omit β-rules and other common rules and focus only on cases of interest.

Firstly, one of the non-conventional features of our system lies in the stratification of results in plain results and cast results:

$$\frac{\Gamma \vdash_{v} v_1 : T \to \underline{C} \qquad \Gamma \vdash_{v} v_2 : T}{\Gamma \vdash_{c} v_1\ v_2 : \underline{C}} \qquad \frac{\Gamma \vdash_{v} v : T \qquad \Gamma, x : T \vdash_{c} c : \underline{C}}{\Gamma \vdash_{c} \mathtt{let}\ x = v\ \mathtt{in}\ c : \underline{C}}$$

$$\frac{\Gamma \vdash_{c} v : T}{\Gamma \vdash_{c} \mathtt{return}\ v : T\ !\ \emptyset} \qquad \frac{\Gamma \vdash_{c} c_1 : T_1\ !\ \Delta \qquad \Gamma, x : T_1 \vdash_{c} c_2 : T_2\ !\ \Delta}{\Gamma \vdash_{c} \mathtt{do}\ x \leftarrow c_1; c_2 : T_2\ !\ \Delta}$$

$$\frac{(\mathtt{Op} : T_1 \to T_2) \in \Sigma \qquad \Gamma \vdash_{v} v : T_1 \qquad \Gamma, y : T_2 \vdash_{c} c : T\ !\ \Delta \qquad \mathtt{Op} \in \Delta}{\Gamma \vdash_{c} \mathtt{Op}\ v\ (y : T_2.c) : T\ !\ \Delta}$$

$$\frac{\Gamma \vdash_{v} v : \underline{C}_1 \Rrightarrow \underline{C}_2 \qquad \Gamma \vdash_{c} c : \underline{C}_1}{\Gamma \vdash_{c} \mathtt{handle}\ c\ \mathtt{with}\ v : \underline{C}_2} \qquad \frac{\Gamma \vdash_{c} c : \underline{C}_1 \qquad \Gamma \vdash_{co} \gamma : \underline{C}_1 \leqslant \underline{C}_2}{\Gamma \vdash_{c} c \rhd \gamma : \underline{C}_2}$$

Fig. 6. ExEff Computation Typing

$$\begin{array}{rrl}
\text{terminal value } v^{T} & ::= & \mathtt{unit} \mid h \mid \mathtt{fun}\ x : T \mapsto c \mid \Lambda\alpha : \tau.v \mid \Lambda\delta.v \mid \lambda\omega : \pi.v \\
\text{value result } v^{R} & ::= & v^{T} \mid v^{T} \rhd \gamma \\
\text{computation result } c^{R} & ::= & \mathtt{return}\ v^{T} \mid (\mathtt{return}\ v^{T}) \rhd \gamma \mid \mathtt{Op}\ v^{R}\ (y : T.c)
\end{array}$$

Terminal values v^T represent conventional values, and value results v^R can either be plain terminal values v^T or terminal values with a cast: $v^T \rhd \gamma$. The same applies to computation results c^R.[5]

Although unusual, this stratification can also be found in Crary's coercion calculus for inclusive subtyping [4], and, more recently, in System F$_C$ [25]. Stratification is crucial for ensuring type preservation. Consider for example the expression

[5] Observe that operation values do not feature an outermost cast operation, as the coercion can always be pushed into its continuation.

($\texttt{return}\ 5 \rhd \langle \texttt{int} \rangle\,!\,\emptyset_{\{\texttt{Op}\}}$), of type $\texttt{int}\,!\,\{\texttt{Op}\}$. We can not reduce the expression further without losing effect information; removing the cast would result in computation ($\texttt{return}\ 5$), of type $\texttt{int}\,!\,\emptyset$. Even if we consider type preservation only up to subtyping, the redex may still occur as a subterm in a context that expects solely the larger type.

Secondly, we need to make sure that casts do not stand in the way of evaluation. This is captured in the so-called "push" rules, all of which appear in Fig. 7.

In relation $v \rightsquigarrow_{\mathrm{v}} v'$, the first rule groups nested casts into a single cast, by means of transitivity. The next three rules capture the essence of push rules: whenever a redex is "blocked" due to a cast, we take the coercion apart and redistribute it (in a type-preserving manner) over the subterms, so that evaluation can progress.

The situation in relation $c \rightsquigarrow_{\mathrm{c}} c'$ is quite similar. The first rule uses transitivity to group nested casts into a single cast. The second rule is a push rule for β-reduction. The third rule pushes a cast out of a `return`-computation. The fourth rule pushes a coercion inside an operation-computation, illustrating why the syntax for c^R does not require casts on operation-computations. The fifth rule is a push rule for sequencing computations and performs two tasks at once. Since we know that the computation bound to x calls no operations, we (a) safely "drop" the impure part of γ, and (b) substitute x with v^T, cast with the pure part of γ (so that types are preserved). The sixth rule handles operation calls in sequencing computations. If an operation is called in a sequencing computation, evaluation is suspended and the rest of the computation is captured in the continuation.

The last four rules are concerned with effect handling. The first of them pushes a coercion on the handler "outwards", such that the handler can be exposed and evaluation is not stuck (similarly to the push rule for term application). The second rule behaves similarly to the push/beta rule for sequencing computations. Finally, the last two rules are concerned with handling of operations. The first of the two captures cases where the called operation is handled by the handler, in which case the respective clause of the handler is called. As illustrated by the rule, like Pretnar [20], ExEff features *deep handlers*: the continuation is also wrapped within a `with-handle` construct. The last rule captures cases where the operation is not covered by the handler and thus remains unhandled.

We have shown that ExEff is type safe:

Theorem 1 (Type Safety)

- *If $\Gamma \vdash_{\mathrm{v}} v : T$ then either v is a result value or $v \rightsquigarrow_{\mathrm{v}} v'$ and $\Gamma \vdash_{\mathrm{v}} v' : T$.*
- *If $\Gamma \vdash_{\mathrm{c}} c : \underline{C}$ then either c is a result computation or $c \rightsquigarrow_{\mathrm{c}} c'$ and $\Gamma \vdash_{\mathrm{c}} c' : \underline{C}$.*

$\boxed{v \rightsquigarrow_{\mathrm{v}} v'}$ **Values**

$$(v^T \rhd \gamma_1) \rhd \gamma_2 \rightsquigarrow_{\mathrm{v}} v^T \rhd (\gamma_1 \gg \gamma_2) \qquad (v^T \rhd \gamma)\ T \rightsquigarrow_{\mathrm{v}} (v^T\ T) \rhd \gamma[T]$$

$$(v^T \rhd \gamma)\ \Delta \rightsquigarrow_{\mathrm{v}} (v^T\ \Delta) \rhd \gamma[\Delta] \qquad (v^T \rhd \gamma_1)\ \gamma_2 \rightsquigarrow_{\mathrm{v}} (v^T\ \gamma_2) \rhd \gamma_1 @ \gamma_2$$

$\boxed{c \rightsquigarrow_{\mathrm{c}} c'}$ **Computations**

$$(c^R \rhd \gamma_1) \rhd \gamma_2 \rightsquigarrow_{\mathrm{c}} c^R \rhd (\gamma_1 \gg \gamma_2) \qquad (v_1^T \rhd \gamma)\ v_2 \rightsquigarrow_{\mathrm{c}} (v_1^T\ (v_2 \rhd \mathit{left}(\gamma))) \rhd \mathit{right}(\gamma)$$

$$\mathtt{return}\ (v^T \rhd \gamma) \rightsquigarrow_{\mathrm{c}} (\mathtt{return}\ v^T) \rhd (\gamma\ !\ \emptyset_\emptyset)$$

$$(\mathtt{Op}\ v^R\ (y : T.c)) \rhd \gamma \rightsquigarrow_{\mathrm{c}} \mathtt{Op}\ v^R\ (y : T.(c \rhd \gamma))$$

$$\mathtt{do}\ x \leftarrow ((\mathtt{return}\ v^T) \rhd \gamma); c_2 \rightsquigarrow_{\mathrm{c}} c_2[(v^T \rhd \mathit{pure}(\gamma))/x]$$

$$\mathtt{do}\ x \leftarrow \mathtt{Op}\ v^R\ (y : T.c_1); c_2 \rightsquigarrow_{\mathrm{c}} \mathtt{Op}\ v^R\ (y : T.\mathtt{do}\ x \leftarrow c_1; c_2)$$

$$\mathtt{handle}\ c\ \mathtt{with}\ (v^T \rhd \gamma) \rightsquigarrow_{\mathrm{c}} (\mathtt{handle}\ (c \rhd \mathit{left}(\gamma))\ \mathtt{with}\ v^T) \rhd \mathit{right}(\gamma)$$

$$\mathtt{handle}\ ((\mathtt{return}\ v^T) \rhd \gamma)\ \mathtt{with}\ h \rightsquigarrow_{\mathrm{c}} c_r[v^T \rhd \mathit{pure}(\gamma)/x]$$

$$\mathtt{handle}\ (\mathtt{Op}\ v^R\ (y : T.c))\ \mathtt{with}\ h \rightsquigarrow_{\mathrm{c}} c_{\mathtt{Op}}[v^R/x, (\mathtt{fun}\ (y : T) \mapsto \mathtt{handle}\ c\ \mathtt{with}\ h)/k]$$

$$\mathtt{handle}\ (\mathtt{Op}\ v^R\ (y : T.c))\ \mathtt{with}\ h \rightsquigarrow_{\mathrm{c}} \mathtt{Op}\ v^R\ (y : T.\mathtt{handle}\ c\ \mathtt{with}\ h)$$

Fig. 7. ExEff Operational Semantics (Selected Rules)

5 Type Inference and Elaboration

This section presents the typing-directed elaboration of ImpEff into ExEff. This elaboration makes all the implicit type and effect information explicit, and introduces explicit term-level coercions to witness the use of subtyping.

After covering the declarative specification of this elaboration, we present a constraint-based algorithm to infer ImpEff types and at the same time elaborate into ExEff. This algorithm alternates between two phases: (1) the syntax-directed generation of constraints from the ImpEff term, and (2) solving these constraints.

5.1 Elaboration of ImpEff into ExEff

The grayed parts of Fig. 2 augment the typing rules for ImpEff value and computation terms with typing-directed elaboration to corresponding ExEff terms. The elaboration is mostly straightforward, mapping every ImpEff construct onto its corresponding ExEff construct while adding explicit type annotations to binders in Rules TmTmAbs, TmHandler and TmOp. Implicit appeals to subtyping are turned into explicit casts with coercions in Rules TmCastV and TmCastC. Rule TmLet introduces explicit binders for skeleton, type, and dirt variables, as well as for constraints. These last also introduce coercion variables ω that can be used in casts. The binders are eliminated in rule TmVar by means of explicit application with skeletons, types, dirts and coercions. The coercions are produced by the auxiliary judgement $\Gamma \vdash_{\mathsf{co}} \gamma : \pi$, defined in Fig. 3, which provides a coercion witness for every subtyping proof.

As a sanity check, we have shown that elaboration preserves types.

Theorem 2 (Type Preservation)

- *If* $\Gamma \vdash_v v : A \rightsquigarrow v'$ *then* $elab_\Gamma(\Gamma) \vdash_{\mathsf{v}} v' : elab_S(A)$.
- *If* $\Gamma \vdash_c c : \underline{C} \rightsquigarrow c'$ *then* $elab_\Gamma(\Gamma) \vdash_{\mathsf{c}} c' : elab_{\underline{C}}(\underline{C})$.

Here $elab_\Gamma(\Gamma)$, $elab_S(A)$ and $elab_{\underline{C}}(\underline{C})$ convert ImpEff environments and types into ExEff environments and types.

5.2 Constraint Generation and Elaboration

Constraint generation with elaboration into ExEff is presented in Figs. 8 (values) and 9 (computations). Before going into the details of each, we first introduce the three auxiliary constructs they use.

$$\begin{array}{rl}
\text{constraint set } \mathcal{P}, \mathcal{Q} & ::= \bullet \mid \tau_1 = \tau_2, \mathcal{P} \mid \alpha : \tau, \mathcal{P} \mid \omega : \pi, \mathcal{P} \\
\text{typing environment } \Gamma & ::= \epsilon \mid \Gamma, x : S \\
\text{substitution } \sigma & ::= \bullet \mid \sigma \cdot [\tau/\varsigma] \mid \sigma \cdot [A/\alpha] \mid \sigma \cdot [\Delta/\delta] \mid \sigma \cdot [\gamma/\omega]
\end{array}$$

At the heart of our algorithm are sets $\mathcal{P}$, containing three different kinds of constraints: (a) skeleton equalities of the form $\tau_1 = \tau_2$, (b) skeleton constraints of the form $\alpha : \tau$, and (c) wanted subtyping constraints of the form $\omega : \pi$. The purpose of the first two becomes clear when we discuss constraint solving, in Sect. 5.3. Next, typing environments Γ only contain term variable bindings, while other variables represent unknowns of their sort and may end up being instantiated after constraint solving. Finally, during type inference we compute substitutions σ, for refining as of yet unknown skeletons, types, dirts, and coercions. The last one is essential, since our algorithm simultaneously performs type inference and elaboration into ExEff.

A substitution σ is a solution of the set $\mathcal{P}$, written as $\sigma \models \mathcal{P}$, if we get derivable judgements after applying σ to all constraints in $\mathcal{P}$.

Values. Constraint generation for values takes the form $\mathcal{Q}; \Gamma \vdash_{\mathsf{v}} v : A \mid \mathcal{Q}'; \sigma \rightsquigarrow v'$. It takes as inputs a set of wanted constraints $\mathcal{Q}$, a typing environment Γ, and a ImpEff value v, and produces a value type A, a new set of wanted constraints $\mathcal{Q}'$, a substitution σ, and a ExEff value v'.

Unlike standard HM, our inference algorithm does not keep constraint generation and solving separate. Instead, the two are interleaved, as indicated by

$\boxed{\mathcal{Q}; \Gamma \vdash_{\mathsf{v}} v : A \mid \mathcal{Q}'; \sigma \rightsquigarrow v'}$ **Values**

$$\frac{(x : \forall\bar{\varsigma}.\overline{\alpha : \tau}.\forall\bar{\delta}.\bar{\pi} \Rightarrow A) \in \Gamma \qquad \sigma = [\overline{\varsigma'/\varsigma}, \overline{\alpha'/\alpha}, \overline{\delta'/\delta}]}{\mathcal{Q}; \Gamma \vdash_{\mathsf{v}} x : \sigma(A) \mid \overline{\omega : \sigma(\pi)}, \overline{\alpha' : \sigma(\tau)}, \mathcal{Q}; \bullet \rightsquigarrow x\ \bar{\varsigma}'\ \bar{\alpha}'\ \bar{\delta}'\ \bar{\omega}}$$

$$\frac{}{\mathcal{Q}; \Gamma \vdash_{\mathsf{v}} \mathtt{unit} : \mathtt{Unit} \mid \mathcal{Q}; \bullet \rightsquigarrow \mathtt{unit}}$$

$$\frac{\alpha : \varsigma, \mathcal{Q}; \Gamma, x : \alpha \vdash_{\mathsf{c}} c : \underline{C} \mid \mathcal{Q}'; \sigma \rightsquigarrow c'}{\mathcal{Q}; \Gamma \vdash_{\mathsf{v}} (\mathtt{fun}\ x \mapsto c) : \sigma(\alpha) \to \underline{C} \mid \mathcal{Q}'; \sigma \rightsquigarrow \mathtt{fun}\ x : \sigma(\alpha) \mapsto c'}$$

$$\frac{\begin{array}{l} \qquad \alpha_r : \varsigma_r, \mathcal{Q}; \Gamma, x : \alpha_r \vdash_{\mathsf{c}} c_r : B_r \,!\, \Delta_r \mid \mathcal{Q}_0; \sigma_r \rightsquigarrow c'_r \qquad \sigma^i = \sigma_i \cdot \sigma_{i-1} \cdot \ldots \cdot \sigma_1 \\ \mathtt{Op}_i \in \mathcal{O} : \\ \quad (\mathtt{Op}_i : A_i \to B_i) \in \Sigma \\ \quad \alpha_i : \varsigma_i, \mathcal{Q}_{i-1}; \sigma^{i-1}(\sigma_r(\Gamma)), x : A_i, k : B_i \to \alpha_i \,!\, \delta_i \vdash_{\mathsf{c}} c_{\mathtt{Op}_i} : B_{\mathtt{Op}_i} \,!\, \Delta_{\mathtt{Op}_i} \mid \mathcal{Q}_i; \sigma_i \rightsquigarrow c'_{\mathtt{Op}_i} \\ \mathcal{Q}' = \alpha_{in} : \varsigma_{in}, \alpha_{out} : \varsigma_{out}, \omega_1 : \sigma^n(B_r) \leqslant \alpha_{out}, \omega_2 : \sigma^n(\Delta_r) \leqslant \delta_{out}, \overline{\omega_{3_i} : \sigma^n(B_{\mathtt{Op}_i}) \leqslant \alpha_{out}}^n, \\ \quad \overline{\omega_{4_i} : \sigma^n(\Delta_{\mathtt{Op}_i}) \leqslant \delta_{out}}^n, \overline{\omega_{5_i} : B_i \to \alpha_{out} \,!\, \delta_{out} \leqslant B_i \to \sigma^n(\alpha_i \,!\, \delta_i)}^n, \\ \quad \omega_6 : \alpha_{in} \leqslant \sigma^n(\sigma_r(\alpha_r)), \omega_7 : \delta_{in} \leqslant \delta_{out} \cup \mathcal{O}, \mathcal{Q}_n \\ c_{res} = \{\mathtt{return}\ y : \sigma^n(\sigma_r(\alpha_r)) \mapsto \sigma^n(c'_r)[y \rhd \omega_6/x] \rhd \omega_1 \,!\, \omega_2 \\ \qquad\quad , [\mathtt{Op}_i\ x\ l \mapsto \sigma^n(c'_{\mathtt{Op}_i})[l \rhd \omega_{5_i}/k] \rhd \omega_{3_i} \,!\, \omega_{4_i}]_{\mathtt{Op}_i \in \mathcal{O}}\} \rhd (\langle \alpha_{in} \rangle \,!\, \omega_7 \Rrightarrow \langle \alpha_{out} \rangle \,!\, \langle \delta_{out} \rangle) \end{array}}{\mathcal{Q}; \Gamma \vdash_{\mathsf{v}} \{\mathtt{return}\ x \mapsto c_r, [\mathtt{Op}\ x\ k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in \mathcal{O}}\} : \alpha_{in} \,!\, \delta_{in} \Rrightarrow \alpha_{out} \,!\, \delta_{out} \mid \mathcal{Q}'; (\sigma^n \cdot \sigma_r) \rightsquigarrow c_{res}}$$

Fig. 8. Constraint Generation with Elaboration (Values)

the additional arguments of our relation: (a) constraints $\mathcal{Q}$ are passed around in a stateful manner (i.e., they are input and output), and (b) substitutions σ generated from constraint solving constitute part of the relation output. We discuss the reason for this interleaved approach in Sect. 5.4; we now focus on the algorithm.

The rules are syntax-directed on the input ImpEff value. The first rule handles term variables x: as usual for constraint-based type inference the rule instantiates the polymorphic type $(\forall\bar{\varsigma}.\overline{\alpha : \tau}.\forall\bar{\delta}.\bar{\pi} \Rightarrow A)$ of x with fresh variables; these are placeholders that are determined during constraint solving. Moreover, the rule extends the wanted constraints $\mathcal{P}$ with $\bar{\pi}$, appropriately instantiated. In ExEff, this corresponds to explicit skeleton, type, dirt, and coercion applications.

More interesting is the third rule, for term abstractions. Like in standard Hindley-Damas-Milner [5], it generates a fresh type variable α for the type of the abstracted term variable x. In addition, it generates a fresh skeleton variable ς, to capture the (yet unknown) shape of α.

As explained in detail in Sect. 5.3, the constraint solver instantiates type variables only through their skeletons annotations. Because we want to allow local constraint solving for the body c of the term abstraction the opportunity to produce a substitution σ that instantiates α, we have to pass in the annotation constraint $\alpha : \varsigma$.[6] We apply the resulting substitution σ to the result type $\sigma(\alpha) \to \underline{C}$.[7]

Finally, the fourth rule is concerned with handlers. Since it is the most complex of the rules, we discuss each of its premises separately:

Firstly, we infer a type $B_r \,!\, \Delta_r$ for the right hand side of the `return`-clause. Since α_r is a fresh unification variable, just like for term abstraction we require $\alpha_r : \varsigma_r$, for a fresh skeleton variable ς_r.

Secondly, we check every operation clause in $\mathcal{O}$ in order. For each clause, we generate fresh skeleton, type, and dirt variables (ς_i, α_i, and δ_i), to account for the (yet unknown) result type $\alpha_i \,!\, \delta_i$ of the continuation k, while inferring type $B_{\mathtt{Op}_i} \,!\, \Delta_{\mathtt{Op}_i}$ for the right-hand-side $c_{\mathtt{Op}_i}$.

More interesting is the (final) set of wanted constraints $\mathcal{Q}'$. First, we assign to the handler the overall type

$$\alpha_{in} \,!\, \delta_{in} \Rrightarrow \alpha_{out} \,!\, \delta_{out}$$

where $\varsigma_{in}, \alpha_{in}, \delta_{in}, \varsigma_{out}, \alpha_{out}, \delta_{out}$ are fresh variables of the respective sorts. In turn, we require that (a) the type of the return clause is a subtype of $\alpha_{out} \,!\, \delta_{out}$ (given by the combination of ω_1 and ω_2), (b) the right-hand-side type of each operation clause is a subtype of the overall result type: $\sigma^n(B_{\mathtt{Op}_i} \,!\, \Delta_{\mathtt{Op}_i}) \leqslant \alpha_{out} \,!\, \delta_{out}$ (witnessed by $\omega_{3_i} \,!\, \omega_{4_i}$), (c) the actual types of the continuations $B_i \to \alpha_{out} \,!\, \delta_{out}$ in the operation clauses should be subtypes of their assumed types $B_i \to \sigma^n(\alpha_i \,!\, \delta_i)$ (witnessed by ω_{5_i}). (d) the overall argument type α_{in} is a subtype of the assumed type of x: $\sigma^n(\sigma_r(\alpha_r))$ (witnessed by ω_6), and (e) the input dirt set δ_{in} is a subtype of the resulting dirt set δ_{out}, extended with the handled operations $\mathcal{O}$ (witnessed by ω_7).

All the aforementioned implicit subtyping relations become explicit in the elaborated term c_{res}, via explicit casts.

Computations. The judgement $\mathcal{Q}; \Gamma \vdash_{\mathsf{c}} c : \underline{C} \mid \mathcal{Q}'; \sigma \rightsquigarrow c'$ generates constraints for computations.

The first rule handles term applications of the form $v_1\, v_2$. After inferring a type for each subterm (A_1 for v_1 and A_2 for v_2), we generate the wanted constraint $\sigma_2(A_1) \leqslant A_2 \to \alpha \,!\, \delta$, with fresh type and dirt variables α and δ, respectively. Associated coercion variable ω is then used in the elaborated term to explicitly (up)cast v_1' to the expected type $A_2 \to \alpha \,!\, \delta$.

The third rule handles polymorphic let-bindings. First, we infer a type A for v, as well as wanted constraints $\mathcal{Q}_v$. Then, we simplify wanted constraints $\mathcal{Q}_v$ by means of function `solve` (which we explain in detail in Sect. 5.3 below), obtaining a substitution σ_1' and a set of *residual constraints* $\mathcal{Q}_v'$.

[6] This hints at why we need to pass constraints in a stateful manner.

[7] Though σ refers to ImpEff types, we abuse notation to save clutter and apply it directly to ExEff entities too.

$\boxed{\mathcal{Q};\Gamma \vdash_{\mathsf{c}} c : \underline{C} \mid \mathcal{Q}';\sigma \rightsquigarrow c'}$ **Computations**

$$\frac{\mathcal{Q};\Gamma \vdash_{\mathsf{v}} v_1 : A_1 \mid \mathcal{Q}_1;\sigma_1 \rightsquigarrow v_1' \qquad \mathcal{Q}_1;\sigma_1(\Gamma) \vdash_{\mathsf{v}} v_2 : A_2 \mid \mathcal{Q}_2;\sigma_2 \rightsquigarrow v_2'}{\mathcal{Q};\Gamma \vdash_{\mathsf{c}} v_1\ v_2 : \alpha \,!\, \delta \mid \alpha : \varsigma, \omega : \sigma_2(A_1) \leqslant A_2 \to \alpha \,!\, \delta, \mathcal{Q}_2; (\sigma_2 \cdot \sigma_1) \rightsquigarrow (\sigma_2(v_1') \rhd \omega)\ v_2'}$$

$$\frac{\mathcal{Q};\Gamma \vdash_{\mathsf{v}} v : A \mid \mathcal{Q}';\sigma \rightsquigarrow v'}{\mathcal{Q};\Gamma \vdash_{\mathsf{c}} \mathtt{return}\ v : A \ !\ \emptyset \mid \mathcal{Q}';\sigma \rightsquigarrow \mathtt{return}\ v'}$$

$$\frac{\begin{array}{c}\mathcal{Q};\Gamma \vdash_{\mathsf{v}} v : A \mid \mathcal{Q}_v;\sigma_1 \rightsquigarrow v' \\ \mathtt{solve}(\bullet;\ \bullet;\ \mathcal{Q}_v) = (\sigma_1', \mathcal{Q}_v') \qquad \mathit{split}(\sigma_1'(\sigma_1(\Gamma)), \mathcal{Q}_v', \sigma_1'(A)) = \langle \bar{\varsigma}, \overline{\alpha : \tau}, \bar{\delta}, \overline{\omega : \pi}, \mathcal{Q}_1 \rangle \\ \mathcal{Q}_1; \sigma_1'(\sigma_1(\Gamma)), x : \forall \bar{\varsigma}.\forall \overline{\alpha : \tau}.\forall \bar{\delta}.\bar{\pi} \Rightarrow \sigma_1'(A) \vdash_{\mathsf{c}} c : \underline{C} \mid \mathcal{Q}_2;\sigma_2 \rightsquigarrow c' \\ c_{res} = \mathtt{let}\ x = \sigma_2(\Lambda\bar{\varsigma}.\Lambda\overline{\alpha : \tau}.\Lambda\bar{\delta}.\Lambda(\overline{\omega : \mathit{elab}_\rho(\pi)}).v')\ \mathtt{in}\ c'\end{array}}{\mathcal{Q};\Gamma \vdash_{\mathsf{c}} \mathtt{let}\ x = v\ \mathtt{in}\ c : \underline{C} \mid \mathcal{Q}_2; (\sigma_2 \cdot \sigma_1' \cdot \sigma_1) \rightsquigarrow c_{res}}$$

$$\frac{\begin{array}{cc}\mathcal{Q};\Gamma \vdash_{\mathsf{v}} v : A_1 \mid \mathcal{Q}_1;\sigma_1 \rightsquigarrow v' & \mathcal{Q}_1;\sigma_1(\Gamma), y : B_{\mathtt{Op}} \vdash_{\mathsf{c}} c : A_2 \ !\ \Delta_2 \mid \mathcal{Q}_2;\sigma_2 \rightsquigarrow c' \\ (\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma & c_{res} = \mathtt{Op}\ (\sigma_2(v') \rhd \omega)\ (y : \mathit{elab}_S(B_{\mathtt{Op}}).c')\end{array}}{\mathcal{Q};\Gamma \vdash_{\mathsf{c}} \mathtt{Op}\ v\ (y : B_{\mathtt{Op}}.c) : A_2 \ !\ \{\mathtt{Op}\} \cup \Delta_2 \mid \omega : \sigma_2(A_1) \leqslant A_{\mathtt{Op}}, \mathcal{Q}_2; (\sigma_2 \cdot \sigma_1) \rightsquigarrow c_{res}}$$

$$\frac{\begin{array}{c}\mathcal{Q};\Gamma \vdash_{\mathsf{c}} c_1 : A_1 \,!\, \Delta_1 \mid \mathcal{Q}_1;\sigma_1 \rightsquigarrow c_1' \qquad \mathcal{Q}_1;\sigma_1(\Gamma), x : A_1 \vdash_{\mathsf{c}} c_2 : A_2 \,!\, \Delta_2 \mid \mathcal{Q}_2;\sigma_2 \rightsquigarrow c_2' \\ c_{res} = \mathtt{do}\ x \leftarrow (\sigma_2(c_1') \rhd \langle \sigma_2(A_1) \rangle \,!\, \omega_1); (c_2' \rhd \langle A_2 \rangle \,!\, \omega_2)\end{array}}{\mathcal{Q};\Gamma \vdash_{\mathsf{c}} \mathtt{do}\ x \leftarrow c_1; c_2 : A_2 \ !\ \delta \mid \omega_1 : \sigma_2(\Delta_1) \leqslant \delta, \omega_2 : \Delta_2 \leqslant \delta, \mathcal{Q}_2; (\sigma_2 \cdot \sigma_1) \rightsquigarrow c_{res}}$$

$$\frac{\begin{array}{c}\mathcal{Q};\Gamma \vdash_{\mathsf{v}} v : A_1 \mid \mathcal{Q}_1;\sigma_1 \rightsquigarrow v' \qquad \mathcal{Q}_1;\sigma_1(\Gamma) \vdash_{\mathsf{c}} c : A_2 \,!\, \Delta_2 \mid \mathcal{Q}_2;\sigma_2 \rightsquigarrow c' \\ \mathcal{Q}' = \alpha_1 : \varsigma_1, \alpha_2 : \varsigma_2, \omega_1 : \sigma_2(A_1) \leqslant (\alpha_1 \,!\, \delta_1 \Rrightarrow \alpha_2 \,!\, \delta_2), \omega_2 : A_2 \leqslant \alpha_1, \omega_3 : \Delta_2 \leqslant \delta_1, \mathcal{Q}_2 \\ c_{res} = \mathtt{handle}\ (c' \rhd (\omega_2 \ !\ \omega_3))\ \mathtt{with}\ (\sigma_2(v') \rhd \omega_1)\end{array}}{\mathcal{Q};\Gamma \vdash_{\mathsf{c}} \mathtt{handle}\ c\ \mathtt{with}\ v : \alpha_2 \ !\ \Delta_2 \mid \mathcal{Q}'; (\sigma_2 \cdot \sigma_1) \rightsquigarrow c_{res}}$$

Fig. 9. Constraint Generation with Elaboration (Computations)

Generalization of x's type is performed by auxiliary function *split*, given by the following clause:

$$\frac{\begin{array}{c}\bar{\varsigma} = \{\varsigma \mid (\alpha : \varsigma) \in \mathcal{Q}, \nexists \alpha'.\alpha' \notin \bar{\alpha} \wedge (\alpha' : \varsigma) \in \mathcal{Q}\} \\ \bar{\alpha} = \mathit{fv}_\alpha(\mathcal{Q}) \cup \mathit{fv}_\alpha(A) \setminus \mathit{fv}_\alpha(\Gamma) \qquad \mathcal{Q}_1 = \{(\omega : \pi) \mid (\omega : \pi) \in \mathcal{Q}, \mathit{fv}(\pi) \not\subseteq \mathit{fv}(\Gamma)\} \\ \bar{\delta} = \mathit{fv}_\delta(\mathcal{Q}) \cup \mathit{fv}_\delta(A) \setminus \mathit{fv}_\delta(\Gamma) \qquad \mathcal{Q}_2 = \mathcal{Q} - \mathcal{Q}_1\end{array}}{\mathit{split}(\Gamma, \mathcal{Q}, A) = \langle \bar{\varsigma}, \overline{\alpha : \tau}, \bar{\delta}, \mathcal{Q}_1, \mathcal{Q}_2 \rangle}$$

In essence, *split* generates the type (scheme) of x in parts. Additionally, it computes the subset $\mathcal{Q}_2$ of the input constraints $\mathcal{Q}$ that do not depend on locally-bound variables. Such constraints can be floated "upwards", and are passed as input when inferring a type for c. The remainder of the rule is self-explanatory.

The fourth rule handles operation calls. Observe that in the elaborated term, we upcast the inferred type to match the expected type in the signature.

The fifth rule handles sequences. The requirement that all computations in a do-construct have the same dirt set is expressed in the wanted constraints $\sigma_2(\varDelta_1) \leqslant \delta$ and $\varDelta_2 \leqslant \delta$ (where δ is a fresh dirt variable; the resulting dirt set), witnessed by coercion variables ω_1 and ω_2. Both coercion variables are used in the elaborated term to upcast c_1 and c_2, such that both draw effects from the same dirt set δ.

Finally, the sixth rule is concerned with effect handling. After inferring type A_1 for the handler v, we require that it takes the form of a handler type, witnessed by coercion variable $\omega_1 : \sigma_2(A_1) \leqslant (\alpha_1 \,!\, \delta_1 \Rrightarrow \alpha_2 \,!\, \delta_2)$, for fresh $\alpha_1, \alpha_2, \delta_1, \delta_2$. To ensure that the type $A_2 \,!\, \varDelta_2$ of c matches the expected type, we require that $A_2 \,!\, \varDelta_2 \leqslant \alpha_1 \,!\, \delta_1$. Our syntax does not include coercion variables for computation subtyping; we achieve the same effect by combining $\omega_2 : A_2 \leqslant \alpha_1$ and $\omega_3 : \varDelta_2 \leqslant \delta_1$.

Theorem 3 (Soundness of Inference). *If $\bullet; \varGamma \vdash_{\mathsf{v}} v : A \mid \mathcal{Q}; \sigma \rightsquigarrow v'$ then for any $\sigma' \models \mathcal{Q}$, we have $(\sigma' \cdot \sigma)(\varGamma) \vdash_v v : \sigma'(A) \rightsquigarrow \sigma'(v')$, and analogously for computations.*

Theorem 4 (Completeness of Inference). *If $\varGamma \vdash_v v : A \rightsquigarrow v'$ then we have $\bullet; \varGamma \vdash_{\mathsf{v}} v : A' \mid \mathcal{Q}; \sigma \rightsquigarrow v''$ and there exists $\sigma' \models \mathcal{Q}$ and γ, such that $\sigma'(v'') = v'$ and $\sigma(\varGamma) \vdash_{\mathsf{co}} \gamma : \sigma'(A') \leqslant A$. An analogous statement holds for computations.*

5.3 Constraint Solving

The second phase of our inference-and-elaboration algorithm is the constraint solver. It is defined by the solve function signature:

$$\boxed{\texttt{solve}(\sigma;\, \mathcal{P};\, \mathcal{Q}) = (\sigma',\, \mathcal{P}')}$$

It takes three inputs: the substitution σ accumulated so far, a list of already processed constraints $\mathcal{P}$, and a queue of still to be processed constraints $\mathcal{Q}$. There are two outputs: the substitution σ' that solves the constraints and the residual constraints $\mathcal{P}'$. The substitutions σ and σ' contain four kinds of mappings: $\varsigma \mapsto \tau$, $\alpha \mapsto A$, $\delta \mapsto \varDelta$ and $\omega \to \gamma$ which instantiate respectively skeleton variables, type variables, dirt variables and coercion variables.

Theorem 5 (Correctness of Solving). *For any set $\mathcal{Q}$, the call* solve$(\bullet; \bullet; \mathcal{Q})$ *either results in a failure, in which case $\mathcal{Q}$ has no solutions, or returns $(\sigma, \mathcal{P})$ such that for any $\sigma' \models \mathcal{Q}$, there exists $\sigma'' \models \mathcal{P}$ such that $\sigma' = \sigma'' \cdot \sigma$.*

The solver is invoked with solve$(\bullet;\, \bullet;\, \mathcal{Q})$, to process the constraints $\mathcal{Q}$ generated in the first phase of the algorithm, i.e., with an empty substitution and no processed constraints. The solve function is defined by case analysis on the queue.

Empty Queue. When the queue is empty, all constraints have been processed. What remains are the residual constraints and the solving substitution σ, which are both returned as the result of the solver.

$$\texttt{solve}(\sigma;\, \mathcal{P};\, \bullet) = (\sigma,\, \mathcal{P})$$

Skeleton Equalities. The next set of cases we consider are those where the queue is non-empty and its first element is an equality between skeletons $\tau_1 = \tau_2$. We consider seven possible cases based on the structure of τ_1 and τ_2 that together essentially implement conventional unification as used in Hindley-Milner type inference [5].

```
$\texttt{solve}(\sigma; \mathcal{P}; \tau_1 = \tau_2, \mathcal{Q}) =$
  $\texttt{match}\ \tau_1 = \tau_2\ \texttt{with}$
    $\mid \varsigma = \varsigma \mapsto \texttt{solve}(\sigma;\ \mathcal{P};\ \mathcal{Q})$
    $\mid \varsigma = \tau \mapsto \texttt{if}\ \varsigma \notin \mathit{fv}_\varsigma(\tau)\ \texttt{then let}\ \sigma' = [\tau/\varsigma]\ \texttt{in solve}(\sigma' \cdot \sigma;\ \bullet; \sigma'(\mathcal{Q}, \mathcal{P}))\ \texttt{else fail}$
    $\mid \tau = \varsigma \mapsto \texttt{if}\ \varsigma \notin \mathit{fv}_\varsigma(\tau)\ \texttt{then let}\ \sigma' = [\tau/\varsigma]\ \texttt{in solve}(\sigma' \cdot \sigma;\ \bullet; \sigma'(\mathcal{Q}, \mathcal{P}))\ \texttt{else fail}$
    $\mid \texttt{Unit} = \texttt{Unit} \mapsto \texttt{solve}(\sigma;\ \mathcal{P};\ \mathcal{Q})$
    $\mid (\tau_1 \rightarrow \tau_2) = (\tau_3 \rightarrow \tau_4) \mapsto \texttt{solve}(\sigma;\ \mathcal{P};\ \tau_1 = \tau_3, \tau_2 = \tau_4, \mathcal{Q})$
    $\mid (\tau_1 \Rrightarrow \tau_2) = (\tau_3 \Rrightarrow \tau_4) \mapsto \texttt{solve}(\sigma;\ \mathcal{P};\ \tau_1 = \tau_3, \tau_2 = \tau_4, \mathcal{Q})$
    $\mid \texttt{otherwise} \mapsto \texttt{fail}$
```

The first case applies when both skeletons are the same type variable ς. Then the equality trivially holds. Hence we drop it and proceed with solving the remaining constraints. The next two cases apply when either τ_1 or τ_2 is a skeleton variable ς. If the occurs check fails, there is no finite solution and the algorithm signals failure. Otherwise, the constraint is solved by instantiating the ς. This additional substitution is accumulated and applied to all other constraints $\mathcal{P}$, $\mathcal{Q}$. Because the substitution might have modified some of the already processed constraints $\mathcal{P}$, we have to revisit them. Hence, they are all pushed back onto the queue, which is processed recursively.

The next three cases consider three different ways in which the two skeletons can have the same instantiated top-level structure. In those cases the equality is decomposed into equalities on the subterms, which are pushed onto the queue and processed recursively.

The last catch-all case deals with all ways in which the two skeletons can be instantiated to different structures. Then there is no solution.

Skeleton Annotations. The next four cases consider a skeleton annotation $\alpha : \tau$ at the head of the queue, and propagate the skeleton instantiation to the type variable. The first case, where the skeleton is a variable ς, has nothing to do, moves the annotation to the processed constraints and proceeds with the remainder of the queue. In the other three cases, the skeleton is instantiated and the solver instantiates the type variable with the corresponding structure, introducing fresh variables for any subterms. The instantiating substitution is accumulated and applied to the remaining constraints, which are processed recursively.

```
solve($\sigma$; $\mathcal{P}$; $\alpha : \tau, \mathcal{Q}$) =
  match $\tau$ with
   | $\varsigma \mapsto$ solve($\sigma$; $\mathcal{P}, \alpha : \tau$; $\mathcal{Q}$)
   | Unit $\mapsto$ let $\sigma' = $ [Unit/$\alpha$] in solve($\sigma' \cdot \sigma$; $\bullet$; $\sigma'(\mathcal{Q}, \mathcal{P})$)
   | $\tau_1 \to \tau_2 \mapsto$ let $\sigma' = [(\alpha_1^{\tau_1} \to \alpha_2^{\tau_2} \,!\, \delta)/\alpha]$ in solve($\sigma' \cdot \sigma$; $\bullet$; $\alpha_1 : \tau_1, \alpha_2 : \tau_2, \sigma'(\mathcal{Q}, \mathcal{P})$)
   | $\tau_1 \Rrightarrow \tau_2 \mapsto$ let $\sigma' = [(\alpha_1^{\tau_1} \,!\, \delta_1 \Rrightarrow \alpha_2^{\tau_2} \,!\, \delta_2)/\alpha]$ in solve($\sigma' \cdot \sigma$; $\bullet$; $\alpha_1 : \tau_1, \alpha_2 : \tau_2, \sigma'(\mathcal{Q}, \mathcal{P})$)
```

Value Type Subtyping. Next are the cases where a subtyping constraint between two value types $A_1 \leqslant A_2$, with as evidence the coercion variable ω, is at the head of the queue. We consider six different situations.

```
solve($\sigma$; $\mathcal{P}$; $\omega : A_1 \leqslant A_2, \mathcal{Q}$) =
  match $A_1 \leqslant A_2$ with
   | $A \leqslant A \mapsto$ let $T = \mathit{elab}_S(A)$ in solve($[\langle T \rangle/\omega] \cdot \sigma$; $\mathcal{P}$; $\mathcal{Q}$)
   | $\alpha^{\tau_1} \leqslant A \mapsto$ let $\tau_2 = \mathit{skeleton}(A)$ in solve($\sigma$; $\mathcal{P}, \omega : \alpha^{\tau_1} \leqslant A$; $\tau_1 = \tau_2, \mathcal{Q}$)
   | $A \leqslant \alpha^{\tau_1} \mapsto$ let $\tau_2 = \mathit{skeleton}(A)$ in solve($\sigma$; $\mathcal{P}, \omega : A \leqslant \alpha^{\tau_1}$; $\tau_2 = \tau_1, \mathcal{Q}$)
   | $(A_1 \to B_1 \,!\, \Delta_1) \leqslant (A_2 \to B_2 \,!\, \Delta_2) \mapsto$ let $\sigma' = [(\omega_1 \to \omega_2 \,!\, \omega_3)/\omega]$ in
     solve($\sigma' \cdot \sigma$; $\mathcal{P}$; $\omega_1 : A_2 \leqslant A_1, \omega_2 : B_1 \leqslant B_2, \omega_3 : \Delta_1 \leqslant \Delta_2, \mathcal{Q}$)
   | $(A_1 \,!\, \Delta_1 \Rrightarrow A_2 \,!\, \Delta_2) \leqslant (A_3 \,!\, \Delta_3 \Rrightarrow A_4 \,!\, \Delta_4) \mapsto$ let $\sigma' = [(\omega_1 \,!\, \omega_2 \Rrightarrow \omega_3 \,!\, \omega_4)/\omega]$ in
     solve($\sigma' \cdot \sigma$; $\mathcal{P}$; $\omega_1 : A_3 \leqslant A_1, \omega_2 : \Delta_3 \leqslant \Delta_1, \omega_3 : A_2 \leqslant A_4, \omega_4 : \Delta_2 \leqslant \Delta_4, \mathcal{Q}$)
   | otherwise $\mapsto$ fail
```

If the two types are equal, the subtyping holds trivially through reflexivity. The solver thus drops the constraint and instantiates ω with the reflexivity coercion $\langle T \rangle$. Note that each coercion variable only appears in one constraint. So we only accumulate the substitution and do not have to apply it to the other constraints. In the next two cases, one of the two types is a type variable α. Then we move the constraint to the processed set. We also add an equality constraint between the skeletons[8] to the queue. This enforces the invariant that only types with the same skeleton are compared. Through the skeleton equality the type structure (if any) from the type is also transferred to the type variable. The next two cases concern two types with the same top-level instantiation. The solver then decomposes the constraint into constraints on the corresponding subterms and appropriately relates the evidence of the old constraint to the new ones. The final case catches all situations where the two types are instantiated with a different structure and thus there is no solution.
Auxiliary function *skeleton*(A) computes the skeleton of A.

Dirt Subtyping. The final six cases deal with subtyping constraints between dirts.

[8] We implicitly annotate every type variable with its skeleton: α^τ.

```
solve($\sigma$; $\mathcal{P}$; $\omega : \Delta \leqslant \Delta', \mathcal{Q}$) =
  match $\Delta \leqslant \Delta'$ with
   | $\mathcal{O} \cup \delta \leqslant \mathcal{O}' \cup \delta' \mapsto$ if $\mathcal{O} \neq \emptyset$ then let $\sigma' = [((\mathcal{O} \backslash \mathcal{O}') \cup \delta'')/\delta', \mathcal{O} \cup \omega'/\omega]$ in
                                                     solve($\sigma' \cdot \sigma$; $\bullet$; $(\omega' : \delta \leq \sigma'(\Delta')), \sigma'(\mathcal{Q}, \mathcal{P})$)
                                               else solve($\sigma$; $\mathcal{P}, (\omega : \Delta \leqslant \Delta')$; $\mathcal{Q}$)
   | $\emptyset \leqslant \Delta' \mapsto$ solve($[\emptyset_{\Delta'}/\omega] \cdot \sigma$; $\mathcal{P}$; $\mathcal{Q}$)
   | $\delta \leqslant \emptyset \mapsto$ let $\sigma' = [\emptyset/\delta;\ \emptyset_{\emptyset}/\omega]$ in solve($\sigma' \cdot \sigma$; $\bullet$; $\sigma'(\mathcal{Q}, \mathcal{P})$)
   | $\mathcal{O} \cup \delta \leqslant \mathcal{O}' \mapsto$
      if $\mathcal{O} \subseteq \mathcal{O}'$ then let $\sigma' = [\mathcal{O} \cup \omega'/\omega]$ in solve($\sigma' \cdot \sigma$; $\mathcal{P}, (\omega' : \delta \leqslant \mathcal{O}')$; $\mathcal{Q}$) else fail
   | $\mathcal{O} \leqslant \mathcal{O}' \mapsto$ if $\mathcal{O} \subseteq \mathcal{O}'$ then let $\sigma' = [\mathcal{O} \cup \emptyset_{\mathcal{O}' \backslash \mathcal{O}}/\omega]$ in solve($\sigma' \cdot \sigma$; $\mathcal{P}$; $\mathcal{Q}$) else fail
   | $\mathcal{O} \leqslant \mathcal{O}' \cup \delta' \mapsto$ let $\sigma' = [(\mathcal{O} \backslash \mathcal{O}') \cup \delta''/\delta';\ \mathcal{O}' \cup \emptyset_{(\mathcal{O}' \backslash \mathcal{O}) \cup \delta''}/\omega]$ in
                          solve($\sigma' \cdot \sigma$; $\bullet$; $\sigma'(\mathcal{Q}, \mathcal{P})$)
```

If the two dirts are of the general form $\mathcal{O} \cup \delta$ and $\mathcal{O}' \cup \delta'$, we distinguish two subcases. Firstly, if $\mathcal{O}$ is empty, there is nothing to be done and we move the constraint to the processed set. Secondly, if $\mathcal{O}$ is non-empty, we partially instantiate δ' with any of the operations that appear in $\mathcal{O}$ but not in $\mathcal{O}'$. We then drop $\mathcal{O}$ from the constraint, and, after substitution, proceed with processing all constraints. For instance, for $\{\mathtt{Op}_1\} \cup \delta \leqslant \{\mathtt{Op}_2\} \cup \delta'$, we instantiate δ' to $\{\mathtt{Op}_1\} \cup \delta''$—where δ'' is a fresh dirt variable—and proceed with the simplified constraint $\delta \leqslant \{\mathtt{Op}_1, \mathtt{Op}_2\} \cup \delta''$. Note that due to the set semantics of dirts, it is not valid to simplify the above constraint to $\delta \leqslant \{\mathtt{Op}_2\} \cup \delta''$. After all the substitution $[\delta \mapsto \{\mathtt{Op}_1\}, \delta'' \mapsto \emptyset]$ solves the former and the original constraint, but not the latter.

The second case, $\emptyset \leqslant \Delta'$, always holds and is discharged by instantiating ω to $\emptyset_{\Delta'}$. The third case, $\delta \leqslant \emptyset$, has only one solution: $\delta \mapsto \emptyset$ with coercion $\emptyset_{\emptyset}$. The fourth case, $\mathcal{O} \cup \delta \leqslant \mathcal{O}'$, has as many solutions as there are subsets of $\mathcal{O}'$, provided that $\mathcal{O} \subseteq \mathcal{O}'$. We then simplify the constraint to $\delta \leqslant \mathcal{O}'$, which we move to the set of processed constraints. The fifth case, $\mathcal{O} \leqslant \mathcal{O}'$, holds iff $\mathcal{O} \subseteq \mathcal{O}'$. The last case, $\mathcal{O} \leqslant \mathcal{O}' \cup \delta'$, is like the first, but without a dirt variable in the left-hand side. We can satisfy it in a similar fashion, by partially instantiating δ' with $(\mathcal{O} \setminus \mathcal{O}') \cup \delta''$—where δ'' is a fresh dirt variable. Now the constraint is satisfied and can be discarded.

Terms

$$\begin{array}{rl}
\text{value } v ::= & x \mid \mathtt{unit} \mid h \mid \mathtt{fun}\ (x : \tau) \mapsto c \mid \Lambda\varsigma.v \mid v\ \tau \\
\text{handler } h ::= & \{\mathtt{return}\ (x : \tau) \mapsto c_r, \mathtt{Op}_1\, x\, k \mapsto c_{\mathtt{Op}_1}, \ldots, \mathtt{Op}_n\, x\, k \mapsto c_{\mathtt{Op}_n}\} \\
\text{computation } c ::= & v_1\ v_2 \mid \mathtt{let}\ x = v\ \mathtt{in}\ c \mid \mathtt{return}\ v \mid \mathtt{Op}\ v\ (y : \tau.c) \\
\mid & \mathtt{do}\ x \leftarrow c_1; c_2 \mid \mathtt{handle}\ c\ \mathtt{with}\ v
\end{array}$$

Types

$$\text{type } \tau ::= \varsigma \mid \tau_1 \to \tau_2 \mid \tau_1 \Rrightarrow \tau_2 \mid \mathtt{Unit} \mid \forall\varsigma.\tau$$

Fig. 10. SkelEff Syntax

5.4 Discussion

At first glance, the constraint generation algorithm of Sect. 5.2 might seem needlessly complex, due to eager constraint solving for let-generalization. Yet, we want to generalize at local `let`-bound values over both type and skeleton variables,[9] which means that we must solve all equations between skeletons before generalizing. In turn, since skeleton constraints are generated when solving subtyping constraints (Sect. 5.3), all skeleton annotations should be available during constraint solving. This can not be achieved unless the generated constraints are propagated statefully.

6 Erasure of Effect Information from ExEff

6.1 The SkelEff Language

The target of the erasure is SkelEff, which is essentially a copy of ExEff from which all effect information Δ, type information T and coercions γ have been removed. Instead, skeletons τ play the role of plain types. Thus, SkelEff is essentially System F extended with term-level (but not type-level) support for algebraic effects. Figure 10 defines the syntax of SkelEff. The type system and operational semantics of SkelEff follow from those of ExEff.

Discussion. The main point of SkelEff is to show that we can erase the effects and subtyping from ExEff to obtain types that are compatible with a System F-like language. At the term-level SkelEff also resembles a subset of Multicore OCaml [6], which provides native support for algebraic effects and handlers but features no explicit polymorphism. Moreover, SkelEff can also serve as a staging area for further elaboration into System F-like languages without support for algebraic effects and handlers (e.g., Haskell or regular OCaml). In those cases, computation terms can be compiled to one of the known encodings in the literature, such as a free monad representation [10,22], with delimited control [11], or using continuation-passing style [13], while values can typically be carried over as they are.

6.2 Erasure

Figure 11 defines erasure functions $\epsilon^{\sigma}_{\mathrm{v}}(v)$, $\epsilon^{\sigma}_{\mathrm{c}}(c)$, $\epsilon^{\sigma}_{\mathrm{V}}(T)$, $\epsilon^{\sigma}_{\mathrm{C}}(\underline{C})$ and $\epsilon^{\sigma}_{\mathrm{E}}(\Gamma)$ for values, computations, value types, computation types, and type environments respectively. All five functions take a substitution σ from the free type variables α to their skeleton τ as an additional parameter.

Thanks to the skeleton-based design of ExEff, erasure is straightforward. All types are erased to their skeletons, dropping quantifiers for type variables and all occurrences of dirt sets. Moreover, coercions are dropped from values

[9] As will become apparent in Sect. 6, if we only generalize at the top over skeleton variables, the erasure does not yield local polymorphism.

$$\begin{aligned}
\epsilon^\sigma_{\mathrm{v}}(x) &= x \\
\epsilon^\sigma_{\mathrm{v}}(\mathtt{unit}) &= \mathtt{unit} \\
\epsilon^\sigma_{\mathrm{v}}(v \rhd \gamma) &= \epsilon^\sigma_{\mathrm{v}}(v) \\
\epsilon^\sigma_{\mathrm{v}}(\mathtt{fun}\ (x : T) \mapsto c) &= \mathtt{fun}\ (x : \epsilon^\sigma_{\mathrm{V}}(T)) \mapsto \epsilon^\sigma_{\mathrm{c}}(c) \\
\epsilon^\sigma_{\mathrm{v}}(\Lambda\varsigma.v) &= \Lambda\varsigma.\epsilon^\sigma_{\mathrm{v}}(v) \\
\epsilon^\sigma_{\mathrm{v}}(\Lambda(\alpha : \tau).v) &= \epsilon^{\sigma\cdot\{\alpha\mapsto\tau\}}_{\mathrm{v}}(v)
\end{aligned}
\qquad
\begin{aligned}
\epsilon^\sigma_{\mathrm{v}}(\Lambda\delta.v) &= \epsilon^\sigma_{\mathrm{v}}(v) \\
\epsilon^\sigma_{\mathrm{v}}(\Lambda(\omega : \pi).v) &= \epsilon^\sigma_{\mathrm{v}}(v) \\
\epsilon^\sigma_{\mathrm{v}}(v\ \tau) &= \epsilon^\sigma_{\mathrm{v}}(v)\ \tau \\
\epsilon^\sigma_{\mathrm{v}}(v\ T) &= \epsilon^\sigma_{\mathrm{v}}(v) \\
\epsilon^\sigma_{\mathrm{v}}(v\ \Delta) &= \epsilon^\sigma_{\mathrm{v}}(v) \\
\epsilon^\sigma_{\mathrm{v}}(v\ \gamma) &= \epsilon^\sigma_{\mathrm{v}}(v)
\end{aligned}$$

$$\begin{aligned}
&\epsilon^\sigma_{\mathrm{v}}(\{\mathtt{return}\ (x : T) \mapsto c_r, [\mathtt{Op}\,x\,k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op}\in\mathcal{O}}\}) = \\
&\quad \{\mathtt{return}\ (x : \epsilon^\sigma_{\mathrm{V}}(T)) \mapsto \epsilon^\sigma_{\mathrm{c}}(c_r), [\mathtt{Op}\,x\,k \mapsto \epsilon^\sigma_{\mathrm{c}}(c_{\mathtt{Op}})]_{\mathtt{Op}\in\mathcal{O}}\}
\end{aligned}$$

$$\begin{aligned}
\epsilon^\sigma_{\mathrm{c}}(v_1\ v_2) &= \epsilon^\sigma_{\mathrm{v}}(v_1)\ \epsilon^\sigma_{\mathrm{v}}(v_2) \\
\epsilon^\sigma_{\mathrm{c}}(\mathtt{let}\ x = v\ \mathtt{in}\ c) &= \mathtt{let}\ x = \epsilon^\sigma_{\mathrm{v}}(v)\ \mathtt{in}\ \epsilon^\sigma_{\mathrm{c}}(c) \\
\epsilon^\sigma_{\mathrm{c}}(\mathtt{return}\ v) &= \mathtt{return}\ (\epsilon^\sigma_{\mathrm{v}}(v)) \\
\epsilon^\sigma_{\mathrm{c}}(\mathtt{Op}\ v\ (y : T.c)) &= \mathtt{Op}\ (\epsilon^\sigma_{\mathrm{v}}(v))\ (y : \epsilon^\sigma_{\mathrm{V}}(T).\epsilon^\sigma_{\mathrm{c}}(c)) \\
\epsilon^\sigma_{\mathrm{c}}(\mathtt{do}\ x \leftarrow c_1; c_2) &= \mathtt{do}\ x \leftarrow \epsilon^\sigma_{\mathrm{c}}(c_1); \epsilon^\sigma_{\mathrm{c}}(c_2) \\
\epsilon^\sigma_{\mathrm{c}}(\mathtt{handle}\ c\ \mathtt{with}\ v) &= \mathtt{handle}\ \epsilon^\sigma_{\mathrm{c}}(c)\ \mathtt{with}\ \epsilon^\sigma_{\mathrm{v}}(v) \\
\epsilon^\sigma_{\mathrm{c}}(c \rhd \gamma) &= \epsilon^\sigma_{\mathrm{c}}(c)
\end{aligned}$$

$$\begin{aligned}
\epsilon^\sigma_{\mathrm{V}}(\alpha) &= \sigma(\alpha) \\
\epsilon^\sigma_{\mathrm{V}}(T \to \underline{C}) &= \epsilon^\sigma_{\mathrm{V}}(T) \to \epsilon^\sigma_{\mathrm{C}}(\underline{C}) \\
\epsilon^\sigma_{\mathrm{V}}(\underline{C}_1 \Rrightarrow \underline{C}_2) &= \epsilon^\sigma_{\mathrm{C}}(\underline{C}_1) \Rrightarrow \epsilon^\sigma_{\mathrm{C}}(\underline{C}_2) \\
\epsilon^\sigma_{\mathrm{V}}(\mathtt{Unit}) &= \mathtt{Unit} \\
\epsilon^\sigma_{\mathrm{V}}(\pi \Rightarrow T) &= \epsilon^\sigma_{\mathrm{V}}(T) \\
\epsilon^\sigma_{\mathrm{V}}(\forall\varsigma.T) &= \forall\varsigma.\epsilon^\sigma_{\mathrm{V}}(T) \\
\epsilon^\sigma_{\mathrm{V}}(\forall(\alpha : \tau).T) &= \epsilon^{\sigma\cdot\{\alpha\mapsto\tau\}}_{\mathrm{V}}(T) \\
\epsilon^\sigma_{\mathrm{V}}(\forall\delta.T) &= \epsilon^\sigma_{\mathrm{V}}(T)
\end{aligned}
\qquad
\begin{aligned}
\epsilon^\sigma_{\mathrm{C}}(T\ !\ \Delta) &= \epsilon^\sigma_{\mathrm{V}}(T) \\
\\
\epsilon^\sigma_{\mathrm{E}}(\epsilon) &= \epsilon \\
\epsilon^\sigma_{\mathrm{E}}(\Gamma, \varsigma) &= \epsilon^\sigma_{\mathrm{E}}(\Gamma), \varsigma \\
\epsilon^\sigma_{\mathrm{E}}(\Gamma, \alpha : \tau) &= \epsilon^{\sigma\cdot\{\alpha\mapsto\tau\}}_{\mathrm{E}}(\Gamma) \\
\epsilon^\sigma_{\mathrm{E}}(\Gamma, \delta) &= \epsilon^\sigma_{\mathrm{E}}(\Gamma) \\
\epsilon^\sigma_{\mathrm{E}}(\Gamma, x : T) &= \epsilon^\sigma_{\mathrm{E}}(\Gamma), x : \epsilon^\sigma_{\mathrm{V}}(T) \\
\epsilon^\sigma_{\mathrm{E}}(\Gamma, \omega : \pi) &= \epsilon^\sigma_{\mathrm{E}}(\Gamma)
\end{aligned}$$

Fig. 11. Definition of type erasure.

and computations. Finally, all binders and elimination forms for type variables, dirt set variables and coercions are dropped from values and type environments.

The expected theorems hold. Firstly, types are preserved by erasure.[10]

Theorem 6 (Type Preservation). *If $\Gamma \vdash_{\mathsf{v}} v : T$ then $\epsilon^{\emptyset}_{\mathrm{E}}(\Gamma) \vdash_{\mathsf{ev}} \epsilon^{\Gamma}_{\mathrm{v}}(v) : \epsilon^{\Gamma}_{\mathrm{V}}(T)$. If $\Gamma \vdash_{\mathsf{c}} c : \underline{C}$ then $\epsilon^{\emptyset}_{\mathrm{E}}(\Gamma) \vdash_{\mathsf{ec}} \epsilon^{\Gamma}_{\mathrm{c}}(c) : \epsilon^{\Gamma}_{\mathrm{C}}(\underline{C})$.*

Here we abuse of notation and use Γ as a substitution from type variables to skeletons used by the erasure functions.

Finally, we have that erasure preserves the operational semantics.

Theorem 7 (Semantic Preservation). *If $v \rightsquigarrow_{\mathrm{v}} v'$ then $\epsilon^\sigma_{\mathrm{v}}(v) \equiv^{\rightsquigarrow}_{\mathrm{v}} \epsilon^\sigma_{\mathrm{v}}(v')$. If $c \rightsquigarrow_{\mathrm{c}} c'$ then $\epsilon^\sigma_{\mathrm{c}}(c) \equiv^{\rightsquigarrow}_{\mathrm{c}} \epsilon^\sigma_{\mathrm{c}}(c')$.*

In both cases, $\equiv^{\rightsquigarrow}$ denotes the congruence closure of the step relation in SkelEff. The choice of substitution σ does not matter as types do not affect the behaviour.

[10] Typing for SkelEff values and computations take the form $\Gamma \vdash_{\mathsf{ev}} v : \tau$ and $\Gamma \vdash_{\mathsf{ec}} c : \tau$.

Discussion. Typically, when type information is erased from call-by-value languages, type binders are erased by replacing them with other (dummy) binders. For instance, the expected definition of erasure would be:

$$\epsilon^{\sigma}_{\mathrm{v}}(\Lambda(\alpha : \tau).v) = \lambda(x : \texttt{Unit}).\epsilon^{\sigma}_{\mathrm{v}}(v)$$

This replacement is motivated by a desire to preserve the behaviour of the typed terms. By dropping binders, values might be turned into computations that trigger their side-effects immediately, rather than at the later point where the original binder was eliminated. However, there is no call for this circumspect approach in our setting, as our grammatical partition of terms in values (without side-effects) and computations (with side-effects) guarantees that this problem cannot happen when we erase values to values and computations to computations.

7 Related Work and Conclusion

Eff's Implicit Type System. The most closely related work is that of Pretnar [20] on inferring algebraic effects for Eff, which is the basis for our implicitly-typed ImpEff calculus, its type system and the type inference algorithm. There are three major differences with Pretnar's inference algorithm.

Firstly, our work introduces an explicitly-typed calculus. For this reason we have extended the constraint generation phase with the elaboration into ExEff and the constraint solving phase with the construction of coercions.

Secondly, we add skeletons to guarantee erasure. Skeletons also allow us to use standard occurs-check during unification. In contrast, unification in Pretnar's algorithm is inspired by Simonet [24] and performs the occurs-check up to the equivalence closure of the subtyping relation. In order to maintain invariants, all variables in an equivalence class (also called a skeleton) must be instantiated simultaneously, whereas we can process one constraint at a time. As these classes turn out to be surrogates for the underlying skeleton types, we have decided to keep the name.

Finally, Pretnar incorporates garbage collection of constraints [19]. The aim of this approach is to obtain unique and simple type schemes by eliminating redundant constraints. Garbage collection is not suitable for our use as type variables and coercions witnessing subtyping constraints cannot simply be dropped, but must be instantiated in a suitable manner, which cannot be done in general.

Consider for instance a situation with type variables α_1, α_2, α_3, α_4, and α_5 where $\alpha_1 \leqslant \alpha_3$, $\alpha_2 \leqslant \alpha_3$, $\alpha_3 \leqslant \alpha_4$, and $\alpha_3 \leqslant \alpha_5$. Suppose that α_3 does not appear in the type. Then garbage collection would eliminate it and replace the constraints by $\alpha_1 \leqslant \alpha_4$, $\alpha_2 \leqslant \alpha_4$, $\alpha_1 \leqslant \alpha_5$, and $\alpha_2 \leqslant \alpha_5$. While garbage collection guarantees that for any ground instantiation of the remaining type variables, there exists a valid ground instantiation for α_3, ExEff would need to be extended with joins (or meets) to express a generically valid instantiation like $\alpha_1 \sqcup \alpha_2$. Moreover, we would need additional coercion formers to establish $\alpha_1 \leqslant (\alpha_1 \sqcup \alpha_2)$ or $(\alpha_1 \sqcup \alpha_2) \leqslant \alpha_4$.

As these additional constructs considerably complicate the calculus, we propose a simpler solution. We use ExEff as it is for internal purposes, but display types to programmers in their garbage-collected form.

Calculi with Explicit Coercions. The notion of explicit coercions is not new; Mitchell [15] introduced the idea of inserting coercions during type inference for ML-based languages, as a means for explicit casting between different numeric types.

Breazu-Tannen et al. [3] also present a translation of languages with inheritance polymorphism into System F, extended with coercions. Although their coercion combinators are very similar to our coercion forms, they do not include inversion forms, which are crucial for the proof of type safety for our system. Moreover, Breazu-Tannen et al.'s coercions are terms, and thus can not be erased.

Much closer to ExEff is Crary's coercion calculus for inclusive subtyping [4], from which we borrowed the stratification of value results. Crary's system supports neither coercion abstraction nor coercion inversion forms.

System F_C [25] uses explicit type-equality coercions to encode complex language features (e.g. GADTs [16] or type families [23]). Though ExEff's coercions are proofs of subtyping rather than type equality, our system has a lot in common with it, including the inversion coercion forms and the "push" rules.

Future Work. Our plans focus on resuming the postponed work on efficient compilation of handlers. First, we intend to adjust program transformations to the explicit type information. We hope that this will not only make the optimizer more robust, but also expose new optimization opportunities. Next, we plan to write compilers to both Multicore OCaml and standard OCaml, though for the latter, we must first adapt the notion of erasure to a target calculus without algebraic effect handlers. Finally, once the compiler shows promising preliminary results, we plan to extend it to other Eff features such as user-defined types or recursion, allowing us to benchmark it on more realistic programs.

Acknowledgements. We would like to thank the anonymous reviewers for careful reading and insightful comments. Part of this work is funded by the Flemish Fund for Scientific Research (FWO). This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

References

1. Barendregt, H.: The Lambda Calculus: Its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics, vol. 3. North-Holland, Amsterdam (1981)
2. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. J. Logic Algebraic Program. **84**(1), 108–123 (2015)
3. Breazu-Tannen, V., Coquand, T., Gunter, C.A., Scedrov, A.: Inheritance as implicit coercion. Inf. Comput. **93**, 172–221 (1991)
4. Crary, K.: Typed compilation of inclusive subtyping. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000, pp, 68–81. ACM, New York (2000)

5. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1982, pp. 207–212. ACM, New York (1982)
6. Dolan, S., White, L., Sivaramakrishnan, K., Yallop, J., Madhavapeddy, A.: Effective concurrency through algebraic effects. In: OCaml Workshop (2015)
7. Girard, J.-Y., Taylor, P., Lafont, Y.: Proofs and Types. Cambridge University Press, Cambridge (1989)
8. Hillerström, D., Lindley, S.: Liberating effects with rows and handlers. In: Chapman, J., Swierstra, W. (eds.) Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, 18 September 2016, pp. 15–27. ACM (2016)
9. Jones, M.P.: A theory of qualified types. In: Krieg-Brückner, B. (ed.) ESOP 1992. LNCS, vol. 582, pp. 287–306. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55253-7_17
10. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming, ICFP 2014, pp. 145–158. ACM (2013)
11. Kiselyov, O., Sivaramakrishnan, K.: Eff directly in OCaml. In: OCaml Workshop (2016)
12. Leijen, D.: Koka: programming with row polymorphic effect types. In: Levy, P., Krishnaswami, N. (eds.) Proceedings of 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014. EPTCS, vol. 153, pp. 100–126 (2014)
13. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017, pp. 486–499. ACM (2017)
14. Lindley, S., McBride, C., McLaughlin, C.: Do be do be do. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017, pp. 500–514. ACM (2017). http://dl.acm.org/citation.cfm?id=3009897
15. Mitchell, J.C.: Coercion and type inference. In: Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1984, pp. 175–185. ACM, New York (1984)
16. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: ICFP 2006 (2006)
17. Plotkin, G.D., Power, J.: Algebraic operations and generic effects. Appl. Categ. Struct. **11**(1), 69–94 (2003)
18. Plotkin, G.D., Pretnar, M.: Handling algebraic effects. Log. Methods Comput. Sci. **9**(4) (2013)
19. Pottier, F.: Simplifying subtyping constraints: a theory. Inf. Comput. **170**(2), 153–183 (2001). https://doi.org/10.1006/inco.2001.2963
20. Pretnar, M.: Inferring algebraic effects. Log. Methods Comput. Sci. **10**(3) (2014)
21. Pretnar, M.: An introduction to algebraic effects and handlers, invited tutorial. Electron. Notes Theoret. Comput. Sci. **319**, 19–35 (2015)
22. Pretnar, M., Saleh, A.H., Faes, A., Schrijvers, T.: Efficient compilation of algebraic effects and handlers. Technical report CW 708, KU Leuven Department of Computer Science (2017)
23. Schrijvers, T., Peyton Jones, S., Chakravarty, M., Sulzmann, M.: Type checking with open type functions. In: ICFP 2008, pp. 51–62. ACM (2008)

24. Simonet, V.: Type inference with structural subtyping: a faithful formalization of an efficient constraint solver. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 283–302. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40018-9_19
25. Sulzmann, M., Chakravarty, M.M.T., Peyton Jones, S., Donnelly, K.: System F with type equality coercions. In: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, TLDI 2007, pp. 53–66. ACM, New York (2007)
26. Wansbrough, K., Peyton Jones, S.L.: Once upon a polymorphic type. In: POPL, pp. 15–28. ACM (1999)

Concurrency

A Separation Logic for a Promising Semantics

Kasper Svendsen[1], Jean Pichon-Pharabod[1], Marko Doko[2](✉), Ori Lahav[3], and Viktor Vafeiadis[2]

[1] University of Cambridge, Cambridge, UK
[2] MPI-SWS, Kaiserslautern and Saarbrücken, Germany
`mdoko@mpi-sws.org`
[3] Tel Aviv University, Tel Aviv, Israel

Abstract. We present SLR, the first expressive program logic for reasoning about concurrent programs under a weak memory model addressing the out-of-thin-air problem. Our logic includes the standard features from existing logics, such as RSL and GPS, that were previously known to be sound only under stronger memory models: (1) separation, (2) per-location invariants, and (3) ownership transfer via release-acquire synchronisation—as well as novel features for reasoning about (4) the absence of out-of-thin-air behaviours and (5) coherence. The logic is proved sound over the recent "promising" memory model of Kang et al., using a substantially different argument to soundness proofs of logics for simpler memory models.

1 Introduction

Recent years have seen the emergence of several program logics [2,6,8,16,23,24, 26–28] for reasoning about programs under weak memory models. These program logics are valuable tools for structuring program correctness proofs, and enabling programmers to reason about the correctness of their programs without necessarily knowing the formal semantics of the programming language. So far, however, they have only been applied to relatively strong memory models (such as TSO [19] or release/acquire consistency [15] that can be expressed as a constraint on individual candidate program executions) and provide little to no reasoning principles to deal with C/C++ "relaxed" accesses.

The main reason for this gap is that the behaviour of relaxed accesses is notoriously hard to specify [3,5]. Up until recently, memory models have either been too strong (e.g., [5,14,17]), forbidding some behaviours observed with modern hardware and compilers, or they have been too weak (e.g., [4]), allowing so-called out-of-thin-air (OOTA) behaviour even though it does not occur in practice and is highly problematic.

One observable behaviour forbidden by the strong models is the load buffering behaviour illustrated by the example below, which, when started with both locations x and y containing 0, can end with both r_1 and r_2 containing 1.

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 357–384, 2018.
https://doi.org/10.1007/978-3-319-89884-1_13

This behaviour is observable on certain ARMv7 processors after the compiler optimises $r_2 + 1 - r_2$ to 1.

$$\begin{array}{l} r_1 := [x]_{\texttt{rlx}};\ \textit{// reads 1} \\ [y]_{\texttt{rlx}} := r_1 \end{array} \Bigg\| \begin{array}{l} r_2 := [y]_{\texttt{rlx}};\ \textit{// reads 1} \\ [x]_{\texttt{rlx}} := r_2 + 1 - r_2 \end{array} \qquad \text{(LB+data+fakedep)}$$

However, one OOTA behaviour they should not allow is the following example by Boehm and Demsky [5]. When started with two completely disjoint lists a and b, by updating them separately in parallel, it should not be allowed to end with a and b pointing to each other, as that would violate physical separation (for simplicity, in these lists, a location just holds the address of the next element):

$$\begin{array}{l} r_1 := [a]_{\texttt{rlx}};\ \textit{// reads } b \\ [r_1]_{\texttt{rlx}} := a \end{array} \Bigg\| \begin{array}{l} r_2 := [b]_{\texttt{rlx}};\ \textit{// reads } a \\ [r_2]_{\texttt{rlx}} := b \end{array} \qquad \text{(Disjoint-Lists)}$$

Because of this specification gap, program logics either do not reason about relaxed accesses, or they assume overly strengthened models that disallow some behaviours that occur in practice (as discussed in Sect. 5).

Recently, there have been several proposals of programming language memory models that allow load buffering behaviour, but forbid obvious out-of-thin-air behaviours [10,13,20]. This development has enabled us to develop a program logic that provides expressive reasoning principles for relaxed accesses, without relying on overly strong models.

In this paper, we present SLR, a separation logic based on RSL [27], extended with strong reasoning principles for relaxed accesses, which we prove sound over the recent "promising" semantics of Kang et al. [13]. SLR features per-location invariants [27] and physical separation [22], as well as novel assertions that we use to show the absence of *OOTA behaviours* and to reason about various *coherence* examples. (Coherence is a property of memory models that requires the existence of a per-location total order on writes that reads respect.)

There are two main contributions of this work.

First, SLR is the first logic which can prove absence of OOTA in all the standard litmus tests. As such, it provides more evidence to the claim that the promising semantics solves the out-of-thin-air problem in a satisfactory way. The paper that introduced the promising semantics [13] comes with three DRF theorems and a simplistic value logic. These reasoning principles are enough to show absence of some simple out-of-thin-air behaviours, but it is still very easy to end up beyond the reasoning power of these two techniques. For instance, they cannot be used to prove that $r_1 = 0$ in the following "random number generator" litmus test[1], where both the x and y locations initially hold 0.

$$\begin{array}{l} r_1 := [x]_{\texttt{rlx}}; \\ [y]_{\texttt{rlx}} := r_1 + 1 \end{array} \Bigg\| \begin{array}{l} r_2 := [y]_{\texttt{rlx}}; \\ [x]_{\texttt{rlx}} := r_2 \end{array} \qquad \text{(RNG)}$$

The subtlety of this litmus test is the following: if the first thread reads a certain value v from x, then it writes $v + 1$ to y, which the second thread can read, and

[1] The litmus test is called this way because some early attempts to solve the OOTA problem allowed this example to return arbitrary values for x and y.

write to x; this, however, does not enable the first thread to read $v + 1$. SLR features novel assertions that allow it to handle those and other examples, as shown in the following section.

The second major contribution is the proof of soundness of SLR over the promising semantics [13][2]. The promising semantics is an operational model that represents memory as a collection of timestamped write messages. Besides the usual steps that execute the next command of a thread, the model has a non-standard step that allows a thread to promise to perform a write in the future, provided that it can guarantee to be able to fulfil its promise. After a write is promised, other threads may read from that write as if it had already happened. Promises allow the load-store reordering needed to exhibit the load buffering behaviour above, and yet seem, from a series of litmus tests, constrained enough so as to not introduce out-of-thin-air behaviour.

Since the promising model is rather different from all other (operational and axiomatic) memory models for which a program logic has been developed, none of the existing approaches for proving soundness of concurrent program logics are applicable to our setting. Two key difficulties in the soundness proof come from dealing with promise steps.

1. Promises are very non-modular, as they can occur at every execution point and can affect locations that may only be accessed much later in the program.
2. Since promised writes can be immediately read by other threads, the soundness proof has to impose the same invariants on promised writes as the ones it imposes on ordinary writes (e.g., that only values satisfying the location's protocol are written). In a logic supporting ownership transfer,[3] however, establishing those invariants is challenging, because a thread may promise to write to x even without having permission to write to x.

To deal with the first challenge, our proof decouples promising steps from ordinary execution steps. We define two semantics of Hoare triples—one "promising", with respect to the full promising semantics, and one "non-promising", with respect to the promising semantics without promising steps—and prove that every Hoare triple that is correct with respect to its non-promising interpretation is also correct with respect to its promising interpretation. This way, we modularise reasoning about promise steps. Even in the non-promising semantics, however, we do allow threads to have outstanding promises. The main difference in the non-promising semantics is that threads are not allowed to issue new promises.

To resolve the second challenge, we observe that in programs verified by SLR, a thread may promise to write to x only if it is able to acquire the necessary write permission before performing the actual write. This follows from promise

[2] As the promising semantics comes with formal proofs of correctness of all the expected local program transformations and of compilation schemes to the x86-TSO, Power, and ARMv8-POP architectures [21], SLR is sound for these architectures too.

[3] Supporting ownership transfer is necessary to provide useful rules for C11 release and acquire accesses.

$$\begin{array}{llll} e \in \mathit{Expr} ::= n & \text{integer} & s \in \mathit{Stm} ::= & \mathsf{\textbf{skip}} \mid s_1; s_2 \mid \mathsf{\textbf{if}}\ e\ \mathsf{\textbf{then}}\ s_1\ \mathsf{\textbf{else}}\ s_2 \\ \quad\quad\quad \mid r & \text{register} & & \mid \mathsf{\textbf{while}}\ e\ \mathsf{\textbf{do}}\ s \mid r := e \mid r := [e]_{\mathtt{rlx}} \\ \quad\quad\quad \mid e_1\ \mathit{op}\ e_2 & \text{arithmetic} & & \mid r := [e]_{\mathtt{acq}} \mid [e_1]_{\mathtt{rlx}} := e_2 \mid [e_1]_{\mathtt{rel}} := e_2 \end{array}$$

Fig. 1. Syntax of the programming language.

certification: the promising semantics requires all promises to be certifiable; that is, for every state of the promising machine, there must exist a non-promising execution of the machine that fulfils all outstanding promises.

We present the SLR assertions and rules informally in Sect. 2. We then give an overview of the promising semantics of Kang et al. [13] in Sect. 3, and use it in Sect. 4 to explain the proof of soundness of SLR. We discuss related work in Sect. 5. *Details of the rules of SLR and its soundness proof can be found in our technical appendix* [1].

2 Our Logic

The novelty of our program logic is to allow non-trivial reasoning about relaxed accesses. Unlike release/acquire accesses, relaxed accesses do not induce synchronisation between threads, so the usual approach of program logics, which relies on ownership transfer, does not apply. Therefore, in addition to reasoning about ownership transfer like a standard separation logic, our logic supports reasoning about relaxed accesses by collecting information about what reads have been observed, and in which order. When combined with information about which writes have been performed, we can deduce that certain executions are impossible.

For concreteness, we consider a minimal "WHILE" programming language with expressions, $e \in \mathit{Expr}$, and statements, $s \in \mathit{Stm}$, whose syntax is given in Fig. 1. Besides local register assignments, statements also include memory reads with relaxed or acquire mode, and memory writes with relaxed or release mode.

2.1 The Assertions of the Logic

The SLR assertion language is generated by the following grammar, where N, l, v, t, π and X all range over a simply-typed term language which we assume includes booleans, locations, values and expressions of the programming language, fractional permissions, and timestamps, and is closed under pairing, finite sets, and sequences. By convention, we assume that l, v, t, π and X range over terms of type location, value, timestamp, permission and sets of pairs of values, and timestamps, respectively.

$$\begin{array}{rl} P, Q \in \mathit{Assn} ::= & \bot \mid \top \mid P \vee Q \mid P \wedge Q \mid P \Rightarrow Q \mid \forall x.\, P \mid \exists x.\, P \mid N_1 = N_2 \mid \phi(N) \\ & \mid P * Q \mid \mathsf{Rel}(l, \phi) \mid \mathsf{Acq}(l, \phi) \mid \mathsf{O}(l, v, t) \mid \mathsf{W}^{\pi}(l, X) \mid \nabla P \end{array}$$

$$\phi \in \mathit{Pred} ::= \lambda x.\, P$$

The grammar contains the standard operators from first order logic and separation logic, the Rel and Acq assertions from RSL [27], and a few novel constructs.

$\mathsf{Rel}(l, \phi)$ grants permission to perform a release write to location l and transfer away the invariant $\phi(v)$, where v is the value written to that location. Conversely, $\mathsf{Acq}(l, \phi)$ grants permission to perform an acquire read from location l and gain access to the invariant $\phi(v)$, where v is the value returned by the read.

The first novel assertion form, $\mathsf{O}(l, v, t)$, records the fact that location l was observed to have value v at timestamp t. The timestamp is used to order it with other reads from the same location. The information this assertion provides is very weak: it merely says that the owner of the assertion has observed that value, it does not imply that any other thread has ever observed it.

The other novel assertion form, $\mathsf{W}^\pi(l, X)$, asserts ownership of location l and records a set of writes X to that location. The fractional permission $\pi \in \mathbb{Q}$ indicates whether ownership is shared or exclusive. Full permission, $\pi = 1$, confers exclusive ownership of location l and ensures that X is the set of all writes to location l; any fraction, $0 < \pi < 1$, confers shared ownership and enforces that X is a lower-bound on the set of writes to location l. The order of writes to l is tracked through timestamps; the set X is thus a set of pairs consisting of the value and the timestamp of the write.

In examples where we only need to refer to the order of writes and not the exact timestamps, we write $\mathsf{W}^\pi(x, \ell)$, where $\ell = [v_1, ..., v_n]$ is a list of values, as shorthand for $\exists t_1, ..., t_n. t_1 > t_2 > \cdots > t_n * \mathsf{W}^\pi(x, \{(v_1, t_1), ..., (v_n, t_n)\})$. The $\mathsf{W}^\pi(x, \ell)$ assertion thus expresses ownership of location x with permission π, and that the writes to x are given by the list ℓ in order, with the most recent write at the front of the list.

Relation Between Reads and Writes. Records of reads and writes can be confronted by the thread owning the exclusive write assertion: all reads must have read values that were written. This is captured formally by the following property:

$$\mathsf{W}^1(x, X) * \mathsf{O}(x, a, t) \Rrightarrow \mathsf{W}^1(x, X) * \mathsf{O}(x, a, t) * (a, t) \in X \quad \text{(Reads-from-Write)}$$

Random Number Generator. These assertions allow us to reason about the "random number generator" litmus test from the Introduction, and to show that it cannot read arbitrarily large values. As discussed in the Introduction, capturing the set of values that are written to x, as made possible by the "invariant-based program logic" of Kang et al. [13, Sect. 5.5] and of Jeffrey and Riley [10, Sect. 6], is not enough, and we make use of our stronger reasoning principles. We use $\mathsf{O}(x, a, t)$ to record what values reads read from each location, and $\mathsf{W}^1(x, \ell)$ to record what sequences of values were written to each location, and then confront these records at the end of the execution. The proof sketch is then as follows:

$$\begin{array}{l||l}
\{\mathsf{W}^1(y, [0]) * \ldots\} & \{\mathsf{W}^1(x, [0]) * \ldots\} \\
r_1 := [x]_{\mathtt{rlx}}; & r_2 := [y]_{\mathtt{rlx}}; \\
\{\mathsf{W}^1(y, [0]) * \mathsf{O}(x, r_1, \_) * \ldots\} & \{\mathsf{W}^1(x, [0]) * \mathsf{O}(y, r_2, \_) * \ldots\} \\
[y]_{\mathtt{rlx}} := r_1 + 1 & [x]_{\mathtt{rlx}} := r_2 \\
\{\mathsf{W}^1(y, [r_1 + 1; 0]) * \mathsf{O}(x, r_1, \_) * \ldots\} & \{\mathsf{W}^1(x, [r_2; 0]) * \mathsf{O}(y, r_2, \_) * \ldots\}
\end{array}$$

At the end of the execution, we are able to draw conclusions about the values of the registers. From $\mathsf{W}^1(x, [r_2; 0])$ and $\mathsf{O}(x, r_1, \_)$, we know that $r_1 \in \{r_2, 0\}$ by rule Reads-from-Write. Similarly, we know that $r_2 \in \{r_1 + 1, 0\}$, and so we can conclude that $r_1 = 0$. We discuss the distribution of resources at the beginning of a program, and their collection at the end of a program, in Theorem 2. Note that we are unable to establish what values the reads read before the end of the litmus test. Indeed, before the end of the execution, nothing enforces that there are no further writes that reads could read from.

2.2 The Rules of the Logic for Relaxed Accesses

We now introduce the rules of our logic by focusing on the rules for *relaxed* accesses. In addition, we support the standard rules from separation logic and Hoare logic, rules for release/acquire accesses (Sect. 2.4), and the following consequence rule:

$$\frac{P \Rrightarrow P' \quad \{P'\}\, c\, \{Q'\} \quad Q' \Rrightarrow Q}{\vdash \{P\}\, c\, \{Q\}} \qquad (\text{CONSEQ})$$

which allows one to use "view shifting" implications to strengthen the precondition and weaken the postcondition.

The rules for relaxed accesses are adapted from the rules of RSL [27] for release/acquire accesses, but use our novel resources to track the more subtle behaviour of relaxed accesses. Since relaxed accesses do not introduce synchronisation, they cannot be used to transfer ownership; they can, however, be used to transfer information. For this reason, as in RSL [27], we associate a predicate ϕ on values to a location x using paired $\mathsf{Rel}(x, \phi)$ and $\mathsf{Acq}(x, \phi)$ resources, for writers and readers, respectively. To write v to x, a writer has to provide $\phi(v)$, and in exchange, when reading v from x, a reader obtains $\phi(v)$. However, here, relaxed writes can only send *pure* predicates (i.e., ones which do not assert ownership of any resources), and relaxed reads can only obtain the assertion from the predicate guarded by a modality ∇[4] that only pure assertions filter through: if P is pure, then $\nabla P \implies P$. All assertions expressible in first-order logic are pure.

Relaxed Write Rule. To write value v (to which the value expression e_2 evaluates) to location x (to which the location expression e_1 evaluates), the thread needs to own a write permission $\mathsf{W}^\pi(x, X)$. Moreover, it needs to provide $\phi(v)$, the assertion associated to the written value, v, to location x by the $\mathsf{Rel}(x, \phi)$ assertion. Because the write is a relaxed write, and therefore does not induce synchronisation, $\phi(v)$ has to be a pure predicate. The write rule updates the record of writes with the value written, timestamped with a timestamp newer than any timestamp for that location that the thread has observed so far; this is expressed by relating it to a previous timestamp that the thread has to provide through an $\mathsf{O}(x, \_, t)$ assertion in the precondition.

[4] This ∇ modality is similar in spirit, but weaker than that of FSL [8].

$$\frac{\phi(v) \text{ is pure}}{\vdash \left\{\begin{matrix} e_1 = x * e_2 = v * \mathsf{W}^\pi(x, X) \\ * \mathsf{Rel}(x, \phi) * \phi(v) * \mathsf{O}(x, \_, t) \end{matrix}\right\} [e_1]_{\mathtt{rlx}} := e_2 \left\{\begin{matrix} \exists t' > t. \\ \mathsf{W}^\pi(x, \{(v, t')\} \cup X) \end{matrix}\right\}} \qquad (\text{W-RLX})$$

The $\mathsf{Rel}(x, \phi)$ assertion is duplicable, so there is no need for the rule to keep it.

In practice, $\mathsf{O}(x, \_, t)$ is taken to be that of the last read from x if it was the last operation on x, and $\mathsf{O}(x, \mathit{fst}(\max(X)), \mathit{snd}(\max(X)))$ if the last operation on x was a write, including the initial write. The latter can be obtained by

$$\mathsf{W}^\pi(x, X) * (v, t) \in X \Rrightarrow \mathsf{W}^\pi(x, X) * \mathsf{O}(x, v, t) \qquad (\text{Write-Observed})$$

Relaxed Read Rule. To read from location x (to which the location expression e evaluates), the thread needs to own an $\mathsf{Acq}(x, \phi)$ assertion, which gives it the right to (almost) obtain assertion $\phi(v)$ upon reading value v from location x. The thread then keeps its $\mathsf{Acq}(x, \phi)$, and obtains an assertion $\mathsf{O}(x, r, t')$ stating that it has read the value now in register r from location x, timestamped with t'. This timestamp is no older than any timestamp for that location that the thread has observed so far, expressed again by relating it to an $\mathsf{O}(x, \_, t)$ assertion in the precondition. Moreover, it obtains the pure portion $\nabla\phi(r)$ of the assertion $\phi(r)$ corresponding to the value read in register r

$$\begin{aligned} \vdash\ & \{e = x * \mathsf{Acq}(x, \phi) * \mathsf{O}(x, \_, t)\} \\ & \quad r := [e]_{\mathtt{rlx}} \\ & \{\exists t' \geq t.\ \mathsf{Acq}(x, \phi) * \mathsf{O}(x, r, t') * \nabla\phi(r)\} \end{aligned} \qquad (\text{R-RLX})$$

Again, we can obtain $\mathsf{O}(x, v_0^x, 0)$, where v_0^x is the initial value of x, from the initial write permission for x, and distribute it to all the threads that will read from x, expressing the fact that the initial value is available to all threads, and use it as the required $\mathsf{O}(x, \_, t)$ in the precondition of the read rule.

Moreover, if a thread owns the exclusive write permission for a location x, then it can take advantage of the fact that it is the only writer at that location to obtain more precise information about its reads from that location: they will read the last value it has written to that location.

$$\begin{aligned} \vdash\ & \{e = x * \mathsf{Acq}(x, \phi) * \mathsf{W}^1(x, X)\} \\ & \quad r := [e]_{\mathtt{rlx}} \\ & \{\exists t.\ (r, t) = \max(X) * \mathsf{Acq}(x, \phi) * \mathsf{W}^1(x, X) * \mathsf{O}(x, r, t) * \nabla\phi(r)\} \end{aligned} \qquad (\text{R-RLX}^*)$$

Separation. With these assertions, we can straightforwardly specify and verify the Disjoint-Lists example. Ownership of an element of a list is simply expressed using a full write permission, $\mathsf{W}^1(x, X)$. This allows including the Disjoint-Lists as a snippet in a larger program where the lists can be shared before or after, and still enforce the separation property we want to establish. While this reasoning sounds underwhelming (and we elide the details), we remark that it is unsound in models that allow OOTA behaviours.

2.3 Reasoning About Coherence

An important feature of many memory models is coherence, that is, the existence of a per-location total order on writes that reads respect. Coherence becomes interesting where there are multiple simultaneous writers to the same location (write/write races). In our logic, write assertions can be split and combined as follows: if $\pi_1 + \pi_2 \leq 1$, $0 < \pi_1$ and $0 < \pi_2$ then

$$\mathsf{W}^{\pi_1+\pi_2}(x, X_1 \cup X_2) \Leftrightarrow \mathsf{W}^{\pi_1}(x, X_1) * \mathsf{W}^{\pi_2}(x, X_2) \qquad \text{(Combine-Writes)}$$

To reason about coherence, the following rules capture the fact that the timestamps of the writes at a given location are all distinct, and totally ordered:

$$\mathsf{W}^{\pi}(x, X) * (v, t) \in X * (v', t') \in X * v \neq v' \Rrightarrow \mathsf{W}^{\pi}(x, X) * t \neq t' \qquad \text{(Different-Writes)}$$

$$\mathsf{W}^{\pi}(x, X) * (\_, t) \in X * (\_, t') \in X \Rrightarrow \mathsf{W}^{\pi}(x, X) * (t < t' \vee t = t' \vee t' < t) \qquad \text{(Writes-Ordered)}$$

CoRR2. One of the basic tests of coherence is the CoRR2 litmus test, which tests whether two threads can disagree on the order of two writes to the same location. The following program, starting with location x holding 0, should not be allowed to finish with $r_1 = 1 * r_2 = 2 * r_3 = 2 * r_4 = 1$, as that would mean that the third thread sees the write of 1 to x before the write of 2 to x, but that the fourth thread sees the write of 2 before the write of 1:

$$[x]_{\mathtt{rlx}} := 1 \;\Big\|\; [x]_{\mathtt{rlx}} := 2 \;\Big\|\; \begin{array}{l} r_1 := [x]_{\mathtt{rlx}}; \\ r_2 := [x]_{\mathtt{rlx}} \end{array} \;\Big\|\; \begin{array}{l} r_3 := [x]_{\mathtt{rlx}}; \\ r_4 := [x]_{\mathtt{rlx}} \end{array} \qquad \text{(CoRR2)}$$

Coherence enforces a total order on the writes to x that is respected by the reads, so if the third thread reads 1 then 2, then the fourth cannot read 2 then 1.

We use the timestamps in the $\mathsf{O}(x, a, t)$ assertions to record the order in which reads read values, and then link the timestamps of the reads with those of the writes. Because we do not transfer anything, the predicate for x is $\lambda v.\, \top$ again, and we elide the associated clutter below.

The proof outline for the writers just records what values have been written:

$$\begin{array}{l} \{\mathsf{W}^{1/2}(x, \{(0,0)\}) * \ldots\} \\ [x]_{\mathtt{rlx}} := 1 \\ \{\exists t_1.\, \mathsf{W}^{1/2}(x, \{(1,t_1),(0,0)\}) * \ldots\} \end{array} \;\Bigg\|\; \begin{array}{l} \{\mathsf{W}^{1/2}(x, \{(0,0)\}) * \ldots\} \\ [x]_{\mathtt{rlx}} := 2 \\ \{\exists t_2.\, \mathsf{W}^{1/2}(x, \{(2,t_2),(0,0)\}) * \ldots\} \end{array}$$

The proof outline for the readers just records what values have been read, and—crucially—in which order.

$$\Bigg\|\; \begin{array}{l} \{\mathsf{Acq}(x, \lambda v.\, \top) * \mathsf{O}(x, 0, 0)\} \\ r_1 := [x]_{\mathtt{rlx}}; \\ \{\exists t_a.\, \mathsf{Acq}(x, \lambda v.\, \top) * \mathsf{O}(x, r_1, t_a) * 0 \leq t_a * \ldots\} \\ r_2 := [x]_{\mathtt{rlx}} \\ \{\exists t_a, t_b.\, \mathsf{O}(x, r_1, t_a) * \mathsf{O}(x, r_2, t_b) * 0 \leq t_a * t_a \leq t_b\} \end{array} \;\Bigg\|\; \begin{array}{l} r_3 := [x]_{\mathtt{rlx}}; \\ r_4 := [x]_{\mathtt{rlx}} \end{array}$$

At the end of the program, by combining the two write permissions using rule Combine-Writes, we obtain $\mathsf{W}^1(x, \{(1, t_1), (2, t_2), (0, 0)\})$. From this, we have $t_1 < t_2$ or $t_2 < t_1$ by rules Different-Writes and Writes-Ordered. Now, assuming $r_1 = 1$ and $r_2 = 2$, we have $t_a < t_b$, and so $t_1 < t_2$ by rule Reads-from-Write. Similarly, assuming $r_3 = 2$ and $r_4 = 1$, we have $t_2 < t_1$. Therefore, we cannot have $r_1 = 1 * r_2 = 2 * r_3 = 2 * r_4 = 1$, so coherence is respected, as desired.

2.4 Handling Release and Acquire Accesses

Next, consider release and acquire accesses, which, in addition to coherence, provide synchronisation and enable the message passing idiom.

$$\begin{array}{l} [x]_{\mathtt{rlx}} := 1; \\ [y]_{\mathtt{rel}} := 1 \end{array} \;\Bigg\|\; \begin{array}{l} r_1 := [y]_{\mathtt{acq}}; \\ \textbf{if } r_1 = 1 \textbf{ then } r_2 := [x]_{\mathtt{rlx}} \end{array} \tag{MP}$$

The first thread writes data (here, 1) to a location x, and signals that the data is ready by writing 1 to a "flag" location y with a release write. The second thread reads the flag location y with an acquire read, and, if it sees that the first thread has signalled that the data has been written, reads the data. The release/acquire pair is sufficient to ensure that the data is then visible to the second thread.

Release/acquire can be understood abstractly in terms of views [15]: a release write contains the view of the writing thread at the time of the writing, and an acquire read updates the view of the reading thread with that of the release write it is reading from. This allows one-way synchronisation of views between threads.

To handle release/acquire accesses in SLR, we can adapt the rules for relaxed accesses by enabling ownership transfer according to predicate associated with the Rel and Acq permissions. The resulting rules are strictly more powerful than the corresponding RSL [27] rules, as they also allow us to reason about coherence.

Release Write Rule. The release write rule is the same as for relaxed writes, but does not require the predicate to be a pure predicate, thereby allowing sending of actual resources, rather than just information:

$$\begin{array}{l} \vdash \; \{e_1 = x * e_2 = v * \mathsf{W}^\pi(x, X) * \mathsf{Rel}(x, \phi) * \phi(v) * \mathsf{O}(x, \_, t)\} \\ \quad [e_1]_{\mathtt{rel}} := e_2 \\ \quad \{\exists t' \geq t.\ \mathsf{W}^\pi(x, \{(v, t')\} \cup X)\} \end{array} \tag{W-REL}$$

Acquire Read Rule. Symmetrically, the acquire read rule is the same as for relaxed reads, but allows the actual resource to be obtained, not just its pure portion:

$$\begin{array}{l} \vdash \; \{e = x * \mathsf{Acq}(x, \phi) * \mathsf{O}(x, \_, t)\} \\ \quad r := [e]_{\mathtt{acq}} \\ \quad \{\exists t' \geq t.\ \mathsf{Acq}(x, \phi[r \mapsto \top]) * \mathsf{O}(x, r, t') * \phi(r)\} \end{array} \tag{R-ACQ}$$

We have to update ϕ to record the fact that we have obtained the resource associated with reading that value, so that we do not erroneously obtain that resource twice; $\phi[v' \mapsto P]$ stands for $\lambda v.\, \textit{if } v = v' \textit{ then } P \textit{ else } \phi(v)$.

As for relaxed accesses, we can strengthen the read rule when the reader is also the exclusive writer to that location:

$$\begin{array}{l} \vdash \ \{\mathsf{Acq}(x,\phi) * \mathsf{W}^1(x,X)\} \\ \quad r := [x]_{\texttt{acq}} \\ \left\{\begin{array}{l} \exists t.\ (r,t) = \max(X) * \mathsf{Acq}(x,\phi[r \mapsto \top]) \\ \qquad * \mathsf{W}^1(x,X) * \mathsf{O}(x,r,t) * \phi(r) \end{array}\right\} \end{array} \tag{R-ACQ*}$$

Additionally, we allow duplicating of release assertions and splitting of acquire assertions, as expressed by the following two rules.

$$\mathsf{Rel}(x,\phi) \Leftrightarrow \mathsf{Rel}(x,\phi) * \mathsf{Rel}(x,\phi) \qquad \text{(Release-Duplicate)}$$

$$\mathsf{Acq}(x, \lambda v.\, \phi_1(v) * \phi_2(v)) \Rrightarrow \mathsf{Acq}(x,\phi_1) * \mathsf{Acq}(x,\phi_2) \qquad \text{(Acquire-Split)}$$

Message Passing. With these rules, we can easify verify the message passing example. Here, we want to transfer a resource from the writer to the reader, namely the state of the data, x. By transferring the write permission for the data to the reader over the "flag" location, y, we allow the reader to use it to read the data precisely. We do that by picking the predicate

$$\phi_y = \lambda v.\ v = 1 \wedge \mathsf{W}^1(x,[1;0]) \vee v \neq 1$$

for y. Since we do not transfer any resource using x, the predicate for x is $\lambda v.\, \top$.

The writer transfers the write permissions for x away on y using ϕ_y:

$$\begin{array}{l} \{\mathsf{W}^1(x,[0]) * \mathsf{Rel}(x,\lambda v.\,\top) * \mathsf{W}^1(y,[0]) * \mathsf{Rel}(y,\phi_y)\} \\ [x]_{\texttt{rlx}} := 1; \\ \{\mathsf{W}^1(x,[1;0]) * \mathsf{W}^1(y,[0]) * \mathsf{Rel}(y,\phi_y)\} \\ \quad \{\mathsf{W}^1(y,\{(0,0)\}) * \mathsf{Rel}(y,\phi_y) * \phi_y(1) * \mathsf{O}(x,0,0)\} \\ [y]_{\texttt{rel}} := 1 \\ \quad \{\exists t_1.\, \mathsf{W}^1(y,\{(1,t_1)\} \cup \{(0,0)\}) * 0 < t_1\} \\ \{\mathsf{W}^1(y,[1;0]) * \mathsf{Rel}(y,\phi_y)\} \end{array}$$

The proof outline for the reader uses the acquire permission ϕ_y for y to obtain $\mathsf{W}^1(x,[1;0])$, which it then uses to know that it reads 1 from x.

$$\left\Vert \begin{array}{l} \{\mathsf{Acq}(y,\phi_y)) * \mathsf{O}(y,0,0) * \mathsf{Acq}(x,\lambda v.\,\top)\} \\ r_1 := [y]_{\texttt{acq}}; \\ \{\exists t_1^y \geq 0.\, \mathsf{Acq}(y,\phi_y[r_1 \mapsto \top]) * \mathsf{O}(y,r_1,t_1^y) * \phi_y(r_1) * \mathsf{Acq}(x,\lambda v.\,\top)\} \\ \{\phi_y(r_1) * \mathsf{Acq}(x,\lambda v.\,\top)\} \\ \textbf{if } r_1 = 1 \textbf{ then} \\ \quad \{\mathsf{W}^1(x,[1;0]) * \mathsf{Acq}(x,\lambda v.\,\top)\} \\ \quad r_2 := [x]_{\texttt{rlx}} \\ \quad \{\mathsf{Acq}(x,\lambda v.\,\top) * \mathsf{W}^1(x,[1;0]) * (r_2 = 1)\} \\ \{r_1 = 1 \Longrightarrow r_2 = 1\} \end{array} \right.$$

2.5 Plain Accesses

Our formal development (in the technical appendix) also features the usual "partial ownership" $x \overset{\pi}{\mapsto} v$ assertion for "plain" (non-atomic) locations, and the usual corresponding rules.

3 The Promising Semantics

In this section, we provide an overview of the promising semantics [13], the model for which we prove SLR sound. Formal details can be found in [1,13].

The promising semantics is an operational semantics that interleaves execution of the threads of a program. Relaxed behaviour is introduced in two ways:

- As in the "strong release/acquire" model [15], the memory is a pool of timestamped messages, and each thread maintains a "view" thereof. A thread may read any value that is not older than the latest value observed by the thread for the given location; in particular, this may well not be the latest value written to that particular location. Timestamps and views model non-multi-copy-atomicity: writes performed by one thread do not become simultaneously visible by all other threads.
- The operational semantics contains a non-standard step: at any point a thread can nondeterministically *promise* a write, provided that, at every point before the write is actually performed, the thread can *certify* the promise, that is, execute the write by running on its own from the current state. Promises are used to enable load-store reordering.

The behaviour of promising steps can be illustrated on the LB+data+fakedep litmus test from the Introduction. The second thread can, at the very start of the execution, promise a write of 1 to x, because it can, by running on its own from the current state, read from y (it will read 0), then write 1 to x (because $0 + 1 - 0 = 1$), thereby fulfilling its promise. On the other hand, the first thread cannot promise a write of 1 to y at the beginning of the execution, because, by running on its own, it can only read 0 from x, and therefore only write 0 to y.

3.1 Storage Subsystem

Formally, the semantics keeps track of writes and promises in a *global configuration*, $gconf = \langle M, P \rangle$, where M is a memory and $P \subseteq M$ is the *promise memory*. We denote by $gconf.\mathtt{M}$ and $gconf.\mathtt{P}$ the components of $gconf$. Both *memories* are finite sets of messages, where a *message* is a tuple $\langle x :_i^o v, R@t] \rangle$, where $x \in Loc$ is the location of the message, $v \in Val$ its value, $i \in Tid$ its originating thread, $t \in Time$ its *timestamp*, R its message *view*, and $o \in \{\mathtt{rlx}, \mathtt{rel}\}$ its message mode, where $Time$ is an infinite set of timestamps, densely totally ordered by $\leq$, with a minimum element, 0. (We return to views later.) We denote $m.\mathtt{loc}$,

$m.\mathtt{val}$, $m.\mathtt{time}$, $m.\mathtt{view}$ and $m.\mathtt{mod}$ the components of a message m. We use the following notation to restrict memories:

$$M(i) \overset{def}{=} \{m \in M \mid m.\mathtt{tid} = i\} \qquad M(\mathtt{rel}) \overset{def}{=} \{m \in M \mid m.\mathtt{mod} = \mathtt{rel}\}$$
$$M(x) \overset{def}{=} \{m \in M \mid m.\mathtt{loc} = x\} \qquad M(\mathtt{rlx}) \overset{def}{=} \{m \in M \mid m.\mathtt{mod} = \mathtt{rlx}\}$$
$$M(i,x) \overset{def}{=} M(i) \cap M(x)$$

A global configuration *gconf* evolves in two ways. First, a message can be "promised" and be added both to *gconf*.M and *gconf*.P. Second, a message can be written, in which case it is either added to *gconf*.M, or removed from *gconf*.P (if it was promised before).

3.2 Thread Subsystem

A *thread state* is a pair $TS = \langle \sigma, V \rangle$, where σ is the internal state of the thread and V is a *view*. We denote by $TS.\sigma$ and $TS.V$ the components of TS.

Thread Internal State. The internal state σ consists of a thread store (denoted $\sigma.\mu$) that assigns values to local registers and a statement to execute (denoted $\sigma.s$). The transitions of the thread internal state are labeled with *memory actions* and are given by an ordinary sequential semantics. As these are routine, we leave their description to the technical appendix.

Views. Thread views are used to enforce coherence, that is, the existence of a per-location total order on writes that reads respect. A view is a function $V : \mathit{Loc} \rightarrow \mathit{Time}$, which records how far the thread has seen in the history of each location. To ensure that a thread does not read stale messages, its view restricts the messages the thread may read, and is increased whenever a thread observes a new message. Messages themselves also carry a view (the thread's view when the message comes from a release write, and the bottom view otherwise) which is incorporated in the thread view when the message is read by an acquire read.

Additional Notations. The order on timestamps, $\leq$, is extended pointwise to views. $\bot$ and $\sqcup$ denote the natural bottom elements and join operations for views. $\{x@t\}$ denotes the view assigning t to x and 0 to other locations.

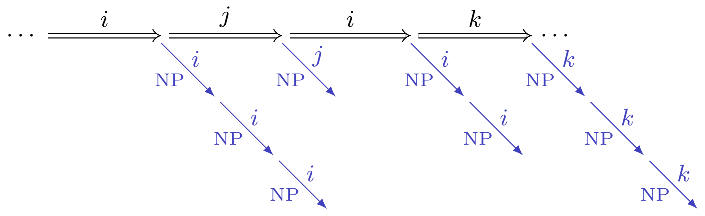
3.3 Interaction Between a Thread and the Storage Subsystem

The interaction between a thread and the storage subsystem is given in terms of transitions of *thread configurations*. Thread configurations are tuples $\langle TS, \langle M, P \rangle \rangle$, where TS is a thread state, and $\langle M, P \rangle$ is a global configuration. These transitions are labelled with $\beta \in \{\mathrm{NP}, \mathrm{prom}\}$ in order to distinguish whether they involve promises or not. A thread can:

- Make an internal transition with no effect on the storage subsystem.
- Read the value v from location x, when there is a matching message in memory that is not outdated according to the thread's view. It then updates its view accordingly: it updates the timestamp for location x and, in addition, incorporates the message view if the read is an acquire read.
- Write the value v to location x. Here, the thread picks a timestamp greater than the one of its current view for the message it adds to memory (or removes from the promise set). If the write is a release write, the message carries the view of the writing thread. Moreover, a release write to x can only be performed when the thread has already fulfilled all its promises to x.
- Non-deterministically promise a relaxed write by adding a message to both M and P.

3.4 Constraining Promises

Now that we have described how threads and promises interact with memory, we can present the certification condition for promises, which is essential to avoid out-of-thin-air behaviours. Accordingly, we define another transition system, $\Longrightarrow$, on top of the previous one, which enforces that the memory remains "consistent", that is, all the promises that have been made can be certified. A thread configuration $\langle TS, \langle M, P\rangle\rangle$ is called *consistent* w.r.t. $i \in Tid$ if thread i can fulfil its promises by executing on its own, or more formally if $\langle TS, \langle M, P\rangle\rangle \xrightarrow{\text{NP}}{}^{*}_{i} \langle TS', \langle M', P'\rangle\rangle$ for some TS', M', P' such that $P'(i) = \emptyset$. Certification is *local*, that is, only thread i is executing during its certification; this is crucial to avoid out-of-thin-air. Further, the certification itself cannot make additional promises, as it is restricted to NP-steps. Here is a visual representation of a promise machine run, together with certifications.



The thread configuration $\Longrightarrow$-transitions allow a thread to (1) take any number of non-promising steps, provided its thread configuration at the end of the sequence of step (intuitively speaking, when it gives control back to the scheduler) is consistent, or (2) take a promising step, again provided that its thread configuration after the step is consistent.

3.5 Full Machine

Finally, the full machine transitions simply lift the thread configuration $\Longrightarrow$-transitions to the machine level. A *machine state* is a tuple $\mathbf{MS} = \langle \mathcal{TS}, \langle M, P\rangle\rangle$, where $\mathcal{TS}$ is a function assigning a thread state TS to every thread, and $\langle M, P\rangle$ is a global configuration. The initial state $\mathbf{MS}^0$ (for a given program) consists of the function $\mathcal{TS}^0$ mapping each thread i to its initial state $\langle \sigma_i^0, \bot\rangle$, where σ_i^0 is the thread's initial local state and $\bot$ is the zero view (all timestamps in views are 0); the initial memory M^0 consisting of one message $\langle x :_0^{\texttt{rlx}} 0, \bot @0]\rangle$ for each location x; and the empty set of promises.

4 Semantics and Soundness

In this section, we present the semantics of SLR, and give a short overview of the soundness proof. Our focus is not on the technical details of the proof, but on the two main challenges in defining the semantics and proving soundness:

1. *Reasoning about promises.* This difficulty arises because promise steps can be nondeterministically performed by the promise machine at any time.
2. *Reasoning about release-acquire ownership transfer in the presence of promises.* The problem is that writes may be promised before the thread has acquired enough resources to allow it to actually perform the write.

4.1 The Intuition

SLR assertions are interpreted by (sets of) *resources*, which represent permissions to write to a certain location and/or to obtain further resources by reading a certain message from memory. As is common in semantics of separation logics, the resources form a partial commutative monoid, and SLR's separating conjunction is interpreted as the composition operation of the monoid.

When defining the meaning of a Hoare triple $\{P\}\ s\ \{Q\}$, we think of the promise machine as if it were manipulating resources: each thread owns some resources and operates using them. The intuitive description of the Hoare triple semantics is that every run of the program s starting from a state containing the resources described by the precondition, P, will be "correct" and, if it terminates, will finish in a state containing the resources described by the postcondition, Q. The notion of a program running correctly can be described in terms of threads "respecting" the resources they own; for example, if a thread is executing a write or fulfilling a promise, it should own a resource representing the write permission.

4.2 A Closer Look at the Resources and the Assertion Semantics

We now take a closer look at the structure of resources and the semantics of assertions, whose formal definitions can be found in Figs. 2 and 3.

The idea is to interpret assertions as predicates over triples consisting of memory, a view, and a resource. We use the resource component to model assertions involving ownership (i.e., write assertions and acquire assertions), and model other assertions using the memory and view components. Once a resource is no longer needed, SLR allows us to drop these from assertions: $P * Q \Rightarrow P$.

To model this we interpret assertions as upwards-closed predicates, that may own more than explicitly asserted. The ordering on memories and views is given by the promising semantics, and the ordering on resources is induced by the composition operation in the resource monoid. For now, we leave the resource composition unspecified, and return to it later.

$$\begin{aligned}
\iota \in \mathit{PredId} &\overset{\mathit{def}}{=} \mathbb{N} \quad \text{(predicate identifiers)}\\
\mathit{Perm} &\overset{\mathit{def}}{=} \{\pi \in \mathbb{Q} \mid 0 \leq \pi \leq 1\} \quad \text{(fractional permissions)}\\
\mathit{Write} &\overset{\mathit{def}}{=} \mathcal{P}(\mathit{Val} \times \mathit{Time})\\
\mathit{WrPerm} &\overset{\mathit{def}}{=} \mathit{Loc} \rightarrow \{(\pi, X) \in \mathit{Perm} \times \mathit{Write} \mid \pi = 0 \Rightarrow X = \emptyset\}\\
\mathit{AcqPerm} &\overset{\mathit{def}}{=} \mathit{Loc} \rightarrow \mathcal{P}(\mathit{PredId})\\
r = (r.\mathtt{wr}, r.\mathtt{acq}) \in \mathit{Res} &\overset{\mathit{def}}{=} \mathit{WrPerm} \times \mathit{AcqPerm} \quad \text{(resources)}\\
\mathcal{W} = (\mathcal{W}.\mathtt{rel}, \mathcal{W}.\mathtt{acq}) \in \mathit{World} &\overset{\mathit{def}}{=} (\mathit{Loc} \rightarrow \mathit{Pred}) \times (\mathit{PredId} \rightharpoonup_{\mathit{fin}} \mathit{Pred}) \quad \text{(worlds)}\\
\mathit{Prop} &\overset{\mathit{def}}{=} \mathit{World} \rightarrow_{\mathit{mon}} \mathcal{P}^{\uparrow}(\mathit{Mem} \times \mathit{View} \times \mathit{Res})
\end{aligned}$$

Fig. 2. Semantic domains used in this section.

In addition, however, we have to deal with assertions that are parametrised by predicates (in our case, $\mathsf{Rel}(x, \phi)$ and $\mathsf{Acq}(x, \phi)$). Doing so is not straightforward because naïve attempts of giving semantics to such assertions result in circular definitions. A common technique for avoiding this circularity is to treat predicates stored in assertions syntactically, and to interpret assertions relative to a *world*, which is used to interpret those syntactic predicates. In our case, worlds consist of two components: the *WrPerm* component associates a syntactic SLR predicate with every location (this component is used to interpret release permissions), while the *AcqPerm* component associates a syntactic predicate with a finite number of currently allocated predicate identifiers (this component is used to interpret acquire permissions). The reason for the more complex structure for acquire permissions is that they can be split (see (Acquire-Split)). Therefore, we allow multiple predicate identifiers associated with a single location. When acquire permissions are divided and split between threads, new predicate identifiers are allocated and associated with predicates in the world. The world ordering, $\mathcal{W}_1 \leq \mathcal{W}_2$, expresses that world $\mathcal{W}_2$ is an extension of $\mathcal{W}_1$ in which new predicate identifiers may have been allocated, but all existing predicate identifiers are associated with the same predicates.

Let us now focus our attention on the assertion semantics. The semantics of assertions, $[\![P]\!]^{\eta}_{\mu}$, is relative to a thread store μ that assigns values to registers, and an environment η that assigns values to logical variables.

The standard logical connectives and quantifiers are interpreted following their usual intuitionistic semantics. The semantics of our novel assertions is given in Fig. 3 and can be explained as follows:

- The observed assertion $\mathsf{O}(x, v, t)$ says that the memory contains a message at location x with value v and timestamp t, and the current thread knows about it (i.e., the thread view contains it).

– The write assertion $\mathsf{W}^{\pi}(x, X)$ asserts ownership of a (partial, with fraction π) write resource at location x, and requires that the largest timestamp recorded in X does not exceed the view of the current thread.
– The acquire assertion, $\mathsf{Acq}(x, \phi)$, asserts that location x has some predicate identifier ι associated with the ϕ predicate in the current world $\mathcal{W}$.
– The release assertion, $\mathsf{Rel}(x, \phi)$, asserts that location x is associated with some predicate ϕ' in the current world such that there exists a syntactic proof of the entailment, $\vdash \forall v.\, \phi(v) \Rightarrow \phi'(v)$. The implication allows us to strengthen the predicate in release assertions.
– Finally, ∇P states that P is satisfiable in the current world.

Note that $\mathsf{W}^{\pi}(x, X)$, $\mathsf{Acq}(x, \phi)$, and $\mathsf{Rel}(x, \phi)$ only talk about owning certain resources, and do not constrain the memory itself at all. In the next subsection, we explain how we relate the abstract resources with the concrete machine state.

$$\begin{aligned}
[\![\mathsf{O}(x, v, t)]\!]^{\eta}_{\mu}(\mathcal{W}) &\overset{def}{=} \{(M, V, r) \mid \\
&\qquad \exists j, R, o.\, \langle [\![x]\!]^{\eta}_{\mu} :^{o}_{j} [\![v]\!]^{\eta}_{\mu}, R@[\![t]\!]^{\eta}_{\mu}] \rangle \in M \wedge [\![t]\!]^{\eta}_{\mu} \leq V(x)\} \\
[\![\mathsf{W}^{\pi}(x, X)]\!]^{\eta}_{\mu}(\mathcal{W}) &\overset{def}{=} \{(M, V, r) \mid \exists \pi' \geq [\![\pi]\!]^{\eta}_{\mu}.\;\; r.\mathtt{wr}([\![x]\!]^{\eta}_{\mu}) = (\pi', [\![X]\!]^{\eta}_{\mu}) \\
&\qquad\qquad \wedge\, \mathit{snd}(\max([\![X]\!]^{\eta}_{\mu})) \leq V([\![x]\!]^{\eta}_{\mu})\} \\
[\![\mathsf{Acq}(x, \phi)]\!]^{\eta}_{\mu}(\mathcal{W}) &\overset{def}{=} \{(M, V, r) \mid \exists \iota \in r.\mathtt{acq}([\![x]\!]^{\eta}_{\mu}).\, \mathcal{W}.\mathtt{acq}(\iota) = \phi\} \\
[\![\mathsf{Rel}(x, \phi)]\!]^{\eta}_{\mu}(\mathcal{W}) &\overset{def}{=} \{(M, V, r) \mid\; \vdash \forall v.\, \phi(v) \Rightarrow \mathcal{W}.\mathtt{rel}([\![x]\!]^{\eta}_{\mu})(v)\} \\
[\![\nabla P]\!]^{\eta}_{\mu}(\mathcal{W}) &\overset{def}{=} \{(M, V, r) \mid [\![P]\!]^{\eta}_{\mu}(\mathcal{W}) \neq \emptyset\}
\end{aligned}$$

Fig. 3. Interpretation of SLR assertions, $[\![\_]\!]^{\eta}_{\mu} \colon \mathit{Assn} \to \mathit{Prop}$

4.3 Relating Concrete State and Resources

Before giving a formal description of the relationship between abstract resources and concrete machine states, we return to the intuition of threads manipulating resources presented in Sect. 4.1.

Consider what happens when a thread executes a release write to a location x. At that point, the thread has to own a release resource represented by $\mathsf{Rel}(x, \phi)$, and to store the value v, it has to own the resources represented by $\phi(v)$. As the write is executed, the thread gives up the ownership of the resources corresponding to $\phi(v)$. Conversely, when a thread that owns the resource represented by $\mathsf{Acq}(x, \phi)$ performs an acquire read of a value v from location x, it will gain ownership of resources satisfying $\phi(v)$. However, this picture does not account for *what happens to the resources that are "in flight"*, i.e., the resources that have been released, but not yet acquired.

Our approach is to associate in-flight resources to messages in the memory. When a thread does a release write, it attaches the resources it released to the message it just added to the memory. That way, a thread performing an acquire read from that message can easily take ownership of the resources that are associated to the message. Formally, as the execution progresses, we update the assignment of resources to messages,

$$u \colon M(\texttt{rel}) \rightarrow (\mathit{PredId} \rightarrow \mathit{Res}).$$

For every release message in memory M, the message resource assignment u gives us a mapping from predicate identifiers to resources. Here, we again use predicate identifiers to be able to track which acquire predicate is being satisfied by which resource. The intended reading of $u(m)(\iota) = r$ is that the resource r attached to the message m satisfies the predicate with the identifier ι.

We also require that the resources attached to a message (i.e., the resources released by the thread that wrote the message) suffice to satisfy all the acquire predicates associated with that particular location. Together, these two properties of our message resource assignment, as formalised in Fig. 4, allow us to describe the release/acquire ownership transfer.

$$\begin{array}{ll}
M \models r, u, \mathcal{W} \overset{\mathit{def}}{=} & \\
\left.\begin{array}{l} \quad \forall m \in M(\texttt{rel}).\ r.\texttt{acq}(m.\texttt{loc}) = \mathit{dom}(u(m)) \\ \land\ \forall \iota \in \mathit{dom}(u(m)). \\ \quad (M, m.\texttt{view}, u(m)(\iota)) \in [\![\mathcal{W}.\texttt{acq}(\iota)(m.\texttt{val})]\!]^{[]}_{[]}(\mathcal{W}) \end{array}\right\} & \begin{array}{c}\text{attached resources}\\ \text{satisfy predicates}\\ \text{they are supposed to}\end{array} \\
\left.\begin{array}{l} \land\ \forall x, v.\ \vdash \mathcal{W}.\texttt{rel}(x)(v) \Rightarrow \circledast_{\iota \in r.\texttt{acq}(x)} \mathcal{W}.\texttt{acq}(\iota)(v) \\ \land\ \forall m \in \mathit{dom}(u).\ \mathit{dom}(u(m)) \subseteq \mathit{dom}(\mathcal{W}.\texttt{acq}) \end{array}\right\} & \begin{array}{c}\text{released resources are}\\ \text{enough to satisfy acquires}\end{array} \\
\left.\begin{array}{l} \land\ \forall m \in M(\texttt{rlx}). \\ \quad (\langle \emptyset, \emptyset \rangle, \lambda x.\, 0, \varepsilon) \in [\![\mathcal{W}.\texttt{rel}(m.\texttt{loc})(m.\texttt{val})]\!]^{[]}_{[]}(\mathcal{W}.\texttt{rel}, []) \end{array}\right\} & \begin{array}{c}\text{no ownership transfer}\\ \text{via relaxed accesses}\end{array}
\end{array}$$

Fig. 4. Message resource satisfaction.

The last condition in the message resource satisfaction relation has to do with relaxed accesses. Since relaxed accesses do not provide synchronisation, we disallow ownership transfer through them. Therefore, we require that the release predicates connected with the relaxed messages are satisfiable with the empty resource. This condition, together with the requirement that the released resources satisfy acquire predicates, forbids ownership transfer via relaxed accesses.

The resource missing from the discussion so far is the write resource (modelling the $\mathsf{W}^{\pi}(x, X)$ assertion). Intuitively, we would like to have the following property: whenever a thread adds a message to the memory, it has to own the corresponding write resource. Recall there are two ways a thread can produce a new message:

1. *A thread performs a write.* This is the straightforward case: we simply require the thread to own the write resource and to update the set of value-timestamp pairs recorded in the resource accordingly.
2. *A thread promises a write.* Here the situation is more subtle, because the thread might not own the write resource at the time it is issuing the promise,

but will acquire the appropriate resource by the time it fulfils the promise. So, in order to assert that the promise step respects the resources owned by the thread, we also need to be able to talk about the resources that the thread *can acquire* in the future.

When dealing with the promises, the saving grace comes from the fact that all promises have to be certifiable, i.e., when issuing a promise a thread has to be able to fulfil it without help from other threads.

Intuitively, the existence of a certification run tells us that even though at the moment a thread issues a promise, it might not have the resources necessary to actually perform the corresponding write, the thread should, by running uninterrupted, still be able to obtain the needed resources before it fulfils the promise. This, in turn, tells us that the needed resources have to be already released by the other threads by the time the promise is made: only resources attached to messages in the memory are available to be acquired, and only the thread that made the promise is allowed to run during the certification; therefore all the available resources have already been released.

The above reasoning shows what it means for the promise steps to "respect resources": when promises are issued, the resources currently owned by a thread, together with all the resources it is able to acquire according to the resources it owns and the current assignment of resources to messages, have to contain the appropriate write resource for the write being promised. The notion of "resources a thread is able to acquire" is expressed through the $\mathrm{canAcq}(r, u)$ predicate. $\mathrm{canAcq}(r, u)$ performs a fixpoint calculation: the resources we have (r) allow us to acquire some more resources from the messages in memory (assignment of resources to messages is given by u), which allows us to acquire some more, and so on. Its formal definition can be found in the technical appendix, and hinges on the fact that u precisely tracks which resources satisfy which predicates.

$$r_1 \bullet r_2 \overset{def}{=} (r_1.\mathtt{wr} \bullet_{\mathtt{wr}} r_2.\mathtt{wr}, r_1.\mathtt{acq} \bullet_{\mathtt{acq}} r_2.\mathtt{acq}) \qquad \varepsilon \overset{def}{=} ([], \lambda\_.\, \emptyset)$$

$$f_1 \bullet_{\mathtt{wr}} f_2 \overset{def}{=} \begin{cases} \lambda x.\, (f_1(x).\mathtt{perm} + f_2(x).\mathtt{perm}, f_1(x).\mathtt{msgs} \cup f_2(x).\mathtt{msgs}) \\ \qquad \text{if } f_1(x).\mathtt{perm} + f_2(x).\mathtt{perm} \leq 1 \text{ for all locations } x \\ \text{undefined otherwise} \end{cases}$$

$$g_1 \bullet_{\mathtt{acq}} g_2 \overset{def}{=} \text{if } \forall x.\, g_1(x) \cap g_2(x) = \emptyset \text{ then } \lambda x.\, g_1(x) \cup g_2(x) \text{ else undefined}$$

Fig. 5. Resource composition.

An important element that was omitted from the discussion so far is the definition of the composition in the resource monoid *Res*. The resource composition, defined in Fig. 5, follows the expected notion of per-component composition. The most important feature is in the composition of write resources: a full permission write resource is only composable with the empty write resource.

At this point, we are equipped with all the necessary ingredients to relate abstract states represented by resources to concrete states $\langle M, P\rangle$ (where M is

$$\lfloor r_F, u, \mathcal{W} \rfloor_T \overset{def}{=} \{\langle M, P\rangle \mid \textbf{let } r = \prod_{i \in TId} r_F(i) \bullet \prod_{m \in M} \prod_{\iota \in dom(u(m))} u(m)(\iota) \textbf{ in}$$
$$\begin{array}{ll} (1) & M \models r, u, \mathcal{W} \wedge \\ (2) & \forall x.\, \{(m.\mathtt{val}, m.\mathtt{time}) \mid m \in M(x) \setminus P\} = r.\mathtt{wr}(x).\mathtt{msgs} \wedge \\ (3) & \forall m \in P.\ m.\mathtt{tid} \notin T \Rightarrow \\ & \quad (r_F(m.\mathtt{tid}) \bullet \mathrm{canAcq}(r_F(m.\mathtt{tid}), u)).\mathtt{wr}(m.\mathtt{loc}).\mathtt{perm} > 0\} \end{array}$$

$r_F\colon \mathit{ThreadId} \to \mathit{Res}$ maps threads to the resources they own.
r is the sum of all the resources distributed among the threads and messages.

Fig. 6. Erasure.

memory, and P is the set of promised messages). We define a function, called *erasure*, that given an assignment of resources to threads, $r_F\colon \mathit{ThreadId} \to \mathit{Res}$, an assignment of resources to messages, u, and a world, $\mathcal{W}$, gives us a set of concrete states satisfying the following conditions:

1. Memory M is consistent with respect to the total resource r and the message resource assignment u at world $\mathcal{W}$.
2. The set of *fulfilled* writes to each location x in $\langle M, P\rangle$ must match the set of writes of all write permissions owned by any thread or associated with any messages, when combined.
3. For all unfulfilled promises to a location x by thread i, thread i must currently own or be able to acquire from u at least a shared write permission for x.

Our formal notion of erasure, defined in Fig. 6, has an additional parameter, a set of thread identifiers T. This set allows us to exclude promises of threads T from the requirement of respecting the resources. As we will see in the following subsection, this additional parameter plays a subtle, but key, role in the soundness proof. (The notion of erasure described above corresponds to the case when $T = \emptyset$.)

Note also that the arguments of erasure very precisely account for who owns which part of the total resource. This diverges from the usual approach in separation logic, where we just give the total resource as the argument to the erasure. Our approach is motivated by Lemma 1, which states that a reader that owns the full write resource for location x knows which value it is going to read from x. This is the key lemma in the soundness proof of the (R-RLX*) and (R-ACQ*) rules.

Lemma 1. If $(M, V, r_F(i)) \in [\![\mathsf{W}^1(x, X)]\!]^{\eta}_{\mu}(\mathcal{W})$, and $\langle M, P\rangle \in \lfloor r_F, u, \mathcal{W}\rfloor_{\{i\}}$ then for all messages $m \in M(x) \setminus P(i)$ such that $V(x) \le m.\mathtt{time}$, we have $m.\mathtt{val} = \mathit{fst}(\max(X))$.

Lemma 1 is looking from the perspective of thread i that owns the full write resource for the location x. This is expressed by $(M, V, r_F(i)) \in [\![\mathsf{W}^1(x, X)]\!]^{\eta}_{\mu}(\mathcal{W})$ (recall that $r_F(i)$ are the resources owned by the thread i). Furthermore, the lemma assumes that the concrete state respects the abstract resources, expressed by $\langle M, P\rangle \in \lfloor r_F, u, \mathcal{W}\rfloor_{\{i\}}$. Under these assumptions, the lemma intuitively tells us that the current thread knows which value it will read from x. Formally, the lemma says that all the messages thread i is allowed to read (i.e., messages in the memory that are not outstanding promises of thread i and whose timestamp is greater or equal to the view of thread i) have the value that appears as the maximal element in the set X.

To see why this lemma holds, consider a message $m \in M(x) \setminus P(i)$. If m is an unfulfilled promise by a different thread j, then, by erasure, it follows that j currently owns or can acquire at least a shared write permission for x. However, this is a contradiction, since thread i currently owns the exclusive write permission, and, by erasure, $r_F(i)$ is disjoint from the resources of all other threads and all resources currently associated with messages by u. Hence, m must be a fulfilled write. By erasure, it follows that the set of fulfilled writes to x is given by the combination of all write permissions. Since $r_F(i)$ owns the exclusive write permission, this is just $r_F(i).\texttt{wr}$. Hence, the set of fulfilled writes is X, and the value of the last fulfilled write is $\mathit{fst}(\max(X))$.

Note that in the reasoning above, it is crucial to know which thread and which message owns which resource. Without precisely tracking this information, we would be unable to prove Lemma 1.

4.4 Soundness

Now that we have our notion of erasure, we can proceed to formalise the meaning of triples, and present the key points of the soundness proof.

Recall our intuitive view of Hoare triples saying that the program only makes steps which respect the resources it owns. This notion is formalised using the *safety* predicate: safety (somewhat simplified; we give its formal definition in Fig. 7) states that it is always safe to perform zero steps, and performing $n+1$ steps is safe if the following two conditions hold:

1. If no more steps can be taken, the current state and resources have to satisfy the postcondition B.
2. If we can take a step which takes us from the state $\langle M, P\rangle$ (which respects our current resources r, the assignment of resources to messages u, and world $\mathcal{W}$) to the state $\langle M', P'\rangle$, then

$$
\begin{aligned}
\mathrm{safe}_0(\sigma, B)(\mathcal{W}_1) &\overset{def}{=} \mathit{Mem} \times \mathit{View} \times \mathit{Res} \\
\mathrm{safe}_{n+1}(\sigma, B)(\mathcal{W}_1) &\overset{def}{=} \{(M_1, V_1, r_1) \mid \forall (M, V, r) \geq (M_1, V_1, r_1).\, \forall \mathcal{W} \geq \mathcal{W}_1. \\
&\qquad (\sigma.s = \textsf{skip} \Rightarrow (M, V, r) \in vs(B(\sigma.\mu))(\mathcal{W})) \\
&\qquad \wedge\ (\forall P, r_F, \sigma', M', P', V', u, i.\ \langle M, P\rangle \in \lfloor r_F[i \mapsto r], u, \mathcal{W}\rfloor_{\emptyset} \wedge \\
&\qquad\quad \langle\langle \sigma, V\rangle, \langle M, P\rangle\rangle \Longrightarrow_i \langle\langle \sigma', V'\rangle, \langle M', P'\rangle\rangle \\
&\qquad\quad \Rightarrow \exists r', u', \mathcal{W}' \geq \mathcal{W}.\, \langle M', P'\rangle \in \lfloor r_F[i \mapsto r'], u', \mathcal{W}'\rfloor_{\emptyset} \wedge \\
&\qquad\qquad (\langle M', P'\rangle, V', r') \in \mathrm{safe}_n(\sigma', B)(\mathcal{W}'))\}
\end{aligned}
$$

Fig. 7. Safety.

(a) there exist resources r', an assignment of resources to messages u', and a future world $\mathcal{W}'$, such that $\langle M', P' \rangle$ respects r', u', and $\mathcal{W}'$, and
(b) we are safe for n more steps starting in the state $\langle M', P' \rangle$ with resources given by r', u' and $\mathcal{W}'$.

Note the following:

– Upon termination, we are not required to satisfy exactly the postcondition B, but its *view shift*. A view shift is a standard notion in concurrent separation logics, which allows updates of the abstract resources which do not affect the concrete state. In our case, this means that resource r can be view-shifted into r' satisfying B as long as the erasure is unchanged. The formal definition of view shifts is given in the appendix.
– Again as is standard in separation logics, safety requires framed resources to be preserved. This is the role of r_F in the safety definition. Frame preservation allows us to compose safety of threads that own compatible resources. However, departing from the standard notion of frame preservation, we precisely track who owns which resource in the frame, because this is important for erasure.

The semantics of Hoare triples is simply defined in terms of the safety predicate. The triple $\{P\}\ s\ \{Q\}$ holds if every logical state satisfying the precondition is safe for any number of steps:

$$[\![\vdash \{P\}\ s\ \{Q\}]\!] \overset{def}{=} \forall n, \mu, \eta, \mathcal{W}.\ [\![P]\!]^{\eta}_{\mu}(\mathcal{W}) \subseteq \mathrm{safe}_n((\mu, s), \lambda\mu'.\ [\![Q]\!]^{\eta}_{\mu'})(\mathcal{W})$$

To establish soundness of the SLR proof rules, we have to prove that the safety predicate holds for arbitrary number of steps, including promise steps. The trouble with reasoning about promise steps is that they can nondeterministically appear at any point of the execution. Therefore, we have to account for them in the soundness proof of every rule of our logic. To make this task manageable, we encapsulate reasoning about the promise steps in a theorem, thus enabling the proofs of soundness for proof rules to consider only the non-promise steps.

To do so, once again certification runs for promises play a pivotal role. Recall that whenever a thread makes a step, it has to be able to fulfil its promises without help from other threads (Sect. 3.4). Since there will be no interference by other threads, performing promise steps during certification is of no use (because promises can only be used by other threads). Therefore, we can assume that the certification runs are always promise-free.

Now that we have noted that certifications are promise-free, the key idea behind encapsulating the reasoning about promises is as follows. If we know that all executions of our program are safe for arbitrarily many non-promising steps, we can use this to conclude that they are safe for promising steps too. Here, we use the fact that certification runs are possible runs of the program, and the fact that certifications are promise-free.

Let us now formalise our key idea. First, we need a way to state that executions are safe for non-promising steps. This is expressed by the *non-promising* *safety* predicate defined in Fig. 8. What we want to conclude is that non-promising safety is enough to establish safety, as expressed by Theorem 1:

Theorem 1 (Non-promising safety implies safety)

$$\forall n, \sigma, B, \mathcal{W}.\ \mathrm{npsafe}_{(n+1,0)}(\sigma, B)(\mathcal{W}) \subseteq \mathrm{safe}_n(\sigma, B)(\mathcal{W})$$

We now discuss several important points in the definition of non-promising safety which enable us to prove this theorem.

Non-promising Safety is Indexed by Pairs of Natural Numbers. When proving Theorem 1, we use promise-free certification runs to establish the safety of the promise steps. A problem we face here is that the length of certification runs is unbounded. Somehow, we have to know that whenever the thread makes a step, it is npsafe for arbitrarily many steps. Our solution is to have npsafe transfinitely indexed over pairs of natural numbers ordered lexicographically. That way, if we are npsafe at index $(n+1, 0)$ and we take a step, we know that we are npsafe at index (n, m) for every m. We are then free to choose a sufficiently large m depending on the length of the certification run we are considering.

Non-promising Safety Considers Configurations that May Contain Promises. It is important to note that the definition of non-promising safety does not require that there are no promises in the starting configuration. The only thing that is required is that no more promises are going to be issued. This is very important for Theorem 1, since safety considers all possible starting configurations (including the ones with existing promises), and if we want the lemma to hold, non-promising safety has to consider all possible starting configurations too.

Erasure Used in the Non-promising Safety does not Constrain Promises of the Current Thread. Non-promising safety does not require promises by the thread being reduced (i.e., thread i) to respect resources. Thus, when reasoning about non-promising safety of thread i, we cannot assume that existing promises by thread i respect resources, but crucially we also do not have to worry about recertifying thread i's promises. However, since the $\xrightarrow{\mathrm{NP}}$ reduction does not recertify promises, we explicitly require that the promises are well formed (via $\mathrm{wf}_{\mathrm{prom}}$ predicate) in order to ensure that we still only consider executions where threads do not read from their own promises.

Additional Constraints by the Non-promising Safety. Non-promising safety also imposes additional constraints on the reducing thread i. In particular, any write permissions owned or acquirable by i after the reduction were already owned or acquirable by i before the reduction step. Intuitively, this holds because thread i can only transfer away resources and take ownership of resources it was already allowed to acquire before reducing. Lastly, non-promising safety requires that if the reduction of i performs any new writes or fulfils any old promises, it must own the write permission for the location of the given message. Together, these two conditions ensure that if a promise is fulfilled during a thread-local certification

$$\begin{aligned}
\text{npsafe}_{(0,m)}(\sigma, B)(\mathcal{W}) &\overset{def}{=} \mathit{Mem} \times \mathit{View} \times \mathit{Res} \\
\text{npsafe}_{(n+1,0)}(\sigma, B)(\mathcal{W}) &\overset{def}{=} \bigcap_{m\in\mathbb{N}} \text{npsafe}_{(n,m)}(\sigma, B)(\mathcal{W}) \\
\text{npsafe}_{(n+1,m+1)}(\sigma, B)(\mathcal{W}_1) &\overset{def}{=} \{(M_1, V_1, r_1) \mid \forall (M, V, r) \geq (M_1, V_1, r_1).\, \forall \mathcal{W} \geq \mathcal{W}_1.
\end{aligned}$$

$$\begin{aligned}
&(\sigma.s = \textsf{skip} \Rightarrow (M, V, r) \in vs(B(\sigma.\mu))(\mathcal{W})) \\
&\wedge (\forall P, r_F, f, \sigma', M', P', V', u, i. \\
&\quad \langle M, P\rangle \in \lfloor r_F[i \mapsto r \bullet f], u, \mathcal{W}\rfloor_{\{i\}} \quad \text{(weak erasure)} \\
&\quad \wedge \quad \langle\langle \sigma, V\rangle, \langle M, P\rangle\rangle \xrightarrow{\text{NP}}_i \langle\langle \sigma', V'\rangle, \langle M', P'\rangle\rangle \quad \text{(only non-promising steps allowed)} \\
&\quad \wedge \text{wf}_{\text{prom}}(P(i), V) \wedge \text{wf}_{\text{prom}}(P'(i), V') \quad \text{(promises well formed)} \\
&\quad \Rightarrow \exists r', u', \mathcal{W}' \geq \mathcal{W}.\, M' \in \lfloor r_F[i \mapsto r' \bullet f], u', \mathcal{W}'\rfloor_{\{i\}} \quad \text{(weak erasure)} \\
&\qquad \wedge (M', V', r') \in \text{npsafe}_{(n+1,m)}(\sigma', B)(\mathcal{W}') \\
&\qquad \wedge r' \bullet \text{canAcq}(r', u') \leq_{\text{o}} r \bullet \text{canAcq}(r, u) \quad \text{(no new res. acquirable after taking a step)} \\
&\qquad \wedge \forall m \in (M' \setminus P') \setminus (M \setminus P).\, r.\texttt{wr}(m.\texttt{loc}).\texttt{perm} > 0\} \quad \text{(when performing a write or fulfiling a promise the thread has to own the appropriate write res.)}
\end{aligned}$$

$$r_1 \leq_{\text{o}} r_2 \overset{def}{=} \forall x.\ r_1.\texttt{wr}(x).\texttt{perm} \leq r_2.\texttt{wr}(x).\texttt{perm}$$

$$\text{wf}_{\text{prom}}(P, V) \overset{def}{=} \forall m \in P.\ V(m.\texttt{loc}) < m.\texttt{time}$$

Fig. 8. Non-promising safety.

and the thread satisfies non-promising safety, then the thread already owned or could acquire the write permission for the location of the promise. This is expressed formally in Lemma 2.

Lemma 2. *Assuming that* $(\langle M, P\rangle, V, r) \in \text{npsafe}_{(n+1,k)}(\sigma, B)(W)$ *and* $\langle M, P\rangle \in \lfloor r_F[i \mapsto r \bullet f], u, W\rfloor_{\{i\}}$ *and* $\langle\langle \sigma, V\rangle, \langle M, P\rangle\rangle \xrightarrow{\text{NP}}{}_i^{k} \langle\langle \sigma', V'\rangle, \langle M', P'\rangle\rangle$ *and* $m \in (M' \setminus P') \setminus (M \setminus P)$, *we have* $(r \bullet \text{canAcq}(r, u)).\texttt{wr}(m.\texttt{loc}).\texttt{perm} > 0$.

The intuition for why Lemma 2 holds is that since only thread i executes, we know by the definition of non-promising safety that any write permission owned or acquirable by i when the promise is fulfilled, it already owns or can acquire in the initial state. Furthermore, whenever a promise is fulfilled, the non-promising safety definition explicitly requires ownership of the corresponding write permission. It follows that the thread already owns or can acquire the write permission for the location of the given promise in the initial state.

Lemma 2 gives us exactly the property that we need to reestablish erasure after the operational semantics introduces a new promise. This makes Lemma 2 the key step in the proof of Theorem 1, which allows us to disentangle reasoning about promising steps and normal reduction steps. Theorem 1 tells us that, in order to prove a proof rule sound, it is enough to prove that the non-promising safety holds for arbitrary indices. This liberates us of the cumbersome reasoning about promise steps and allows us to focus on non-promising reduction steps when proving the proof rules sound.

We can now state our top-level correctness theorem, Theorem 2. Since our language only has top-level parallel composition, we need a way to distribute initial resources to the various threads, and to collect all the resources once all the threads have finished. The correctness theorem gives us precisely that:

Theorem 2 (Correctness). *If A is a finite set of locations and*

1. $\vdash \forall x \in A.\, \phi_x(0)$
2. $\vdash \circledast_{x \in A} \mathsf{Rel}(x, \phi_x) * \mathsf{Acq}(x, \phi_x) * \mathsf{W}^1(x, \{(0,0)\}) \Rrightarrow \circledast_{i \in \mathit{Tid}} P_i$
3. $\vdash \{P_i\}\ s_i\ \{Q_i\}$ *for all* i
4. $\langle \lambda i.\, \langle (\mu_i, s_i), \bot \rangle, \langle M^0, \emptyset \rangle \rangle \Longrightarrow^* \langle \mathcal{TS}, \mathit{gconf} \rangle$ *and* $\mathcal{TS}(i).\sigma = \textsf{skip}$ *for all* i
5. $\vdash \circledast_{i \in \mathit{Tid}} Q_i \Rrightarrow Q$
6. $\mathit{FRV}(Q_i) \cap \mathit{FRV}(Q_j) = \emptyset$ *for all distinct* $i, j \in \mathit{Tid}$

then there exist μ, r, and $\mathcal{W}$ such that $(\mathit{gconf}.M, \sqcup_i \mathcal{TS}(i).V, r) \in [\![Q]\!]^{[]}_{\mu}(\mathcal{W})$ and $\forall i \in \mathit{Tid}.\, \forall a \in \mathit{FRV}(Q_i).\, \mu(a) = \mathcal{TS}(i).\mu(a)$, where $\mathit{FRV}(P)$ denotes the set of free register variables in P.

5 Related Work

There are a number of techniques for reasoning under relaxed memory models, but besides the DRF theorems and some simple invariant logics [10,13], no other techniques have been proved sound for a model allowing the weak behaviour of LB+data+fakedep from the introduction. The "invariant-based program logics" are by design unable to reason about programs like the random number generator, where having a bound on the set of values written to a location is not enough, let alone reasoning about functional correctness of a program.

Relaxed Separation Logic (RSL). Among program logics for relaxed memory, the most closely related is RSL [27]. There are two versions of RSL: a weak one that is sound with respect to the C/C++11 memory model, which features out-of-thin-air reads, and a stronger one that is sound with respect to a variant of the C/C++11 memory that forbids load buffering.

The weak version of RSL forbids relaxed writes completely, and does not constrain the value returned by a relaxed read. The stronger version provides single-location invariants for relaxed accesses, but its soundness proof relies strongly on a strengthened version of C/C++11 without $\mathit{po} \cup \mathit{rf}$ cycles (where po is program order, and rf is the reads-from relation), which forbids load buffering.

When it comes to reasoning about coherence properties, even the strong version of RSL is surprisingly weak: it cannot be used to verify any of the coherence examples in this paper. In fact, RSL can be shown sound with respect to much weaker coherence axioms than what C/C++11 relaxed accesses provide.

One notable feature of RSL which we do not support is read-modify-write (RMW) instructions (such as compare-and-swap and fetch-and-add). However, the soundness proof of SLR makes no simplifying assumptions about the promising semantics which would affect the semantics of RMW instructions. Therefore, we are confident that enhancing SLR with rules for RMW instructions would not substantially affect the structure of the soundness proof, presented in Sect. 4.

Other Program Logics. FSL [8] extends (the strong version of) RSL with stronger rules for relaxed accesses in the presence of release/acquire fences. In FSL, a release fence can be used to package an assertion with a modality, which a relaxed write can then transfer. Conversely, the ownership obtained by a relaxed read is guarded by a symmetric modality than needs an acquire fence to be unpacked. The soundness proof of FSL also relies on $po \cup rf$ acyclicity. Moreover, it is known to be unsound in models where load buffering is allowed [9, Sect. 5.2].

A number of other logics—GPS [26], iGPS [12], OGRA [16], iCAP-TSO [24], the rely-guarantee proof system for TSO of Ridge [23], and the program logic for TSO of Wehrman and Berdine [28]—have been developed for even stronger memory models (release/acquire or TSO), and also rely quite strongly on—and try to expose—the stronger consistency guarantees provided by those models.

The framework of Alglave and Cousot [2] for reasoning about relaxed concurrent programs is parametric with respect to an axiomatic "per-execution" memory model. By construction, as argued by Batty et al. [3], such models cannot be used to define a language-level model allowing the weak behaviour of LB+data+fakedep and similar litmus tests while forbidding out-of-thin-air behaviours. Moreover, their framework does not provide the usual abstraction facilities of program logics.

The lace logic of Bornat et al. [6] targets hardware memory models, in particular Power. It relies on annotating the program with "per-execution" constraints, and on syntactic features of the program. For example, it distinguishes LB+data+fakedep from LB+data+po, its variant where the write of second thread is $[x]_{\texttt{rlx}} := 1$, and is thus unsuitable to address out-of-thin-air behaviours.

Other Approaches. Besides program logics, another way to reason about programs under weak memory models is to reduce the act of reasoning under a memory model M to reasoning under a stronger model M'—typically, but not necessarily, sequential consistency [7,18]. One can often establish DRF theorems stating that a program without any races when executed under M' has the same behaviours when executed under M as when executed under M'. For the promising semantics, Kang et al. [13, Sect. 5.4] have established such theorems for M' being release-acquire consistency, sequential consistency, and the promise-free promising semantics, for suitable notions of races. The last one, the "Promise-Free DRF" theorem, is applicable to the Disjoint-Lists program from the introduction, but none of these theorems can be applied to any of the other examples of this paper, as they are racy. Moreover, these theorems are not compositional, as they do not state anything about the Disjoint-Lists program when put inside a larger, racy program—for example, just an extra read of a from another thread.

6 Conclusion

In this paper, we have presented the first expressive logic that is sound under the promising semantics, and have demonstrated its expressiveness with a number of examples. Our logic can be seen both as a general proof technique for reasoning about concurrent programs, and also as tool for proving the absence of out-of-thin-air behaviour for challenging examples, and reasoning about coherence. In the future, we would like to extend the logic to cover more of relaxed memory, more advanced reasoning principles, such as those available in GPS [26], and mechanise its soundness proof.

Interesting aspects of relaxed memory we would like to also cover are read-modify-writes and fences. These would allow us to consider concurrent algorithms like circular buffers and the atomic reference counter verified in FSL++ [9]. This could be done by adapting the corresponding rules of RSL and GPS; moreover, we could adapt them with our new approach to reason about coherence.

To mechanise the soundness proof, we intend to use the Iris framework [11], which has already been used to prove the soundness of iGPS [12], a variant of the GPS program logic. To do this, however, we have to overcome one technical limitation of Iris. Namely, the current version of Iris is step-indexed over $\mathbb{N}$, while our semantics uses transfinite step-indexing over $\mathbb{N} \times \mathbb{N}$ to define non-promising safety and allow us to reason about certifications of arbitrary length for each reduction step. Progress has been made towards transfinitely step-indexed logical relations that may be applicable to a transfinitely step-indexed version of Iris [25].

Acknowledgments. We would like to thank the reviewers for their feedback. The research was supported in part by the Danish Council for Independent Research (project DFF – 4181-00273), by a European Research Council Consolidator Grant for the project "RustBelt" (grant agreement no. 683289), and by Len Blavatnik and the Blavatnik Family foundation.

References

1. Supplementary material for this paper. http://plv.mpi-sws.org/slr/appendix.pdf
2. Alglave, J., Cousot, P.: Ogre and Pythia: an invariance proof method for weak consistency models. In: POPL 2017, pp. 3–18. ACM, New York (2017). http://doi.acm.org/10.1145/2994593
3. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 283–307. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_12
4. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL 2011, pp. 55–66. ACM, New York (2011). http://doi.acm.org/10.1145/1926385.1926394

5. Boehm, H.J., Demsky, B.: Outlawing ghosts: avoiding out-of-thin-air results. In: Proceedings of the Workshop on Memory Systems Performance and Correctness, MSPC 2014, pp. 7:1–7:6. ACM, New York (2014). http://doi.acm.org/10.1145/2618128.2618134
6. Bornat, R., Alglave, J., Parkinson, M.: New lace and arsenic (2016). https://arxiv.org/abs/1512.01416
7. Bouajjani, A., Derevenetc, E., Meyer, R.: Robustness against relaxed memory models. In: Software Engineering 2014, Fachtagung des GI-Fachbereichs Softwaretechnik, 25–28 Februar 2014, Kiel, Deutschland, pp. 85–86 (2014)
8. Doko, M., Vafeiadis, V.: A program logic for C11 memory fences. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 413–430. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_20
9. Doko, M., Vafeiadis, V.: Tackling real-life relaxed concurrency with FSL++. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 448–475. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_17
10. Jeffrey, A., Riely, J.: On thin air reads towards an event structures model of relaxed memory. In: LICS 2016, pp. 759–767. ACM, New York (2016)
11. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up (2017)
12. Kaiser, J.O., Dang, H.H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: reasoning about release-acquire consistency in Iris. In: ECOOP 2017 (2017)
13. Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: POPL 2017. ACM, New York (2017). http://doi.acm.org/10.1145/3009837.3009850
14. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI (2017)
15. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: POPL 2016, pp. 649–662. ACM, New York (2016). http://doi.acm.org/10.1145/2837614.2837643
16. Lahav, O., Vafeiadis, V.: Owicki-gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 311–323. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_25
17. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL, pp. 378–391. ACM, New York (2005)
18. Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-TSO. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 478–503. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_23
19. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_27
20. Pichon-Pharabod, J., Sewell, P.: A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In: POPL 2016, pp. 622–633. ACM, New York (2016)
21. Podkopaev, A., Lahav, O., Vafeiadis, V.: Promising compilation to ARMv8 POP. In: ECOOP 2017. LIPIcs, vol. 74, pp. 22:1–22:28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)

22. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22–25 July 2002, Copenhagen, Denmark, pp. 55–74 (2002). https://doi.org/10.1109/LICS.2002.1029817
23. Ridge, T.: A rely-guarantee proof system for x86-TSO. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 55–70. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15057-9_4
24. Sieczkowski, F., Svendsen, K., Birkedal, L., Pichon-Pharabod, J.: A separation logic for fictional sequential consistency. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 736–761. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_30
25. Svendsen, K., Sieczkowski, F., Birkedal, L.: Transfinite step-indexing: decoupling concrete and logical steps. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 727–751. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_28
26. Turon, A., Vafeiadis, V., Dreyer, D.: GPS: navigating weak memory with ghosts, protocols, and separation. In: 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2014, pp. 691–707. ACM, New York (2014)
27. Vafeiadis, V., Narayan, C.: Relaxed separation logic: a program logic for C11 concurrency. In: OOPSLA 2013, pp. 867–884. ACM, New York (2013)
28. Wehrman, I., Berdine, J.: A proposal for weak-memory local reasoning. In: LOLA (2011)

Logical Reasoning for Disjoint Permissions

Xuan-Bach Le[1(✉)] and Aquinas Hobor[1,2]

[1] National University of Singapore, Singapore, Singapore
bachdylan@gmail.com
[2] Yale-NUS College, Singapore, Singapore

Abstract. Resource sharing is a fundamental phenomenon in concurrent programming where several threads have permissions to access a common resource. Logics for verification need to capture the notion of permission ownership and transfer. One typical practice is the use of rational numbers in $(0, 1]$ as permissions in which 1 is the full permission and the rest are fractional permissions. Rational permissions are not a good fit for separation logic because they remove the essential "disjointness" feature of the logic itself. We propose a general logic framework that supports permission reasoning in separation logic while preserving disjointness. Our framework is applicable to sophisticated verification tasks such as doing induction over the finiteness of the heap within the object logic or carrying out biabductive inference. We can also prove precision of recursive predicates within the object logic. We developed the ShareInfer tool to benchmark our techniques. We introduce "scaling separation algebras," a compositional extension of separation algebras, to model our logic, and use them to construct a concrete model.

1 Introduction

The last 15 years have witnessed great strides in program verification [7,27,39,43,44,46]. One major area of focus has been concurrent programs following Concurrent Separation Logic (CSL) [40]. The key rule of CSL is Parallel:

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \qquad \{P_2\}\ c_2\ \{Q_2\}}{\{P_1 \star P_2\}\ c_1||c_2\ \{Q_1 \star Q_2\}}\ \textsc{Parallel}$$

In this rule, we write $c_1||c_2$ to indicate the parallel execution of commands c_1 and c_2. The separating conjunction $\star$ indicates that the resources used by the threads is disjoint in some useful way, *i.e.* that there are no dangerous races. Many subsequent program logics [18,20,30,31,45] have introduced increasingly sophisticated notions of "resource disjointness" for the Parallel rule.

Fractional permissions (also called "shares") are a relatively simple enhancement to separation logic's original notion of disjointness [4]. Rather than owning a resource (e.g. a memory cell) entirely, a thread is permitted to own a part/fraction of that resource. The more of a resource a thread owns, the more

© The Author(s) 2018
A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 385–414, 2018.
https://doi.org/10.1007/978-3-319-89884-1_14

actions it is permitted to take, a mapping called a *policy*. In this paper we will use the original policy of Bornat [4] to keep the examples straightforward: non-zero ownership of a memory cell permits reading while full ownership also permits writing. More modern logics allow for a variety of more flexible share policies [13,28,42], but our techniques still apply. Fractional permissions are less expressive than the "protocol-based" notions of disjointness used in program logics such as FCSL [38,44], Iris [30], and TaDa [16], but are well-suited for common concurrent programming patterns such as read sharing and so have been incorporated into many program logics and verification tools [19,26,28,31,36,41].

Since fractionals are simpler and more uniform than protocol-based logics, they are amenable to automation [26,33]. However, previous techniques had difficulty with the inductive predicates common in SL proofs. We introduce *predicate multiplication*, a concise method for specifying the fractional sharing of complex predicates, writing $\pi \cdot P$ to indicate that we own the π-share of the arbitrary predicate P, *e.g.* $0.5 \cdot \mathsf{tree}(x)$ indicates a tree rooted at x and we own half of each of the nodes in the tree. If set up properly, predicate multiplication handles inductive predicates smoothly and is well-suited for automation because:

Section 3 it distributes with bientailments—*e.g.* $\pi \cdot (P \wedge Q) \dashv\vdash (\pi \cdot P) \wedge (\pi \cdot Q)$— enabling rewriting techniques and both forwards and backwards reasoning;
Section 4 it works smoothly with the inference process of biabduction [10]; and
Section 5 the side conditions required for bientailments and biabduction can be verified directly in the object logic, leveraging existing entailment checkers.

There has been significant work in recent years on tool support for protocol-based approaches [15,19,29,30,48], but they require significant user input and provide essentially no inference. Fractional permissions and protocol-based approaches are thus complementary: fractionals can handle large amounts of relatively simple concurrent code with minimal user guidance, while protocol-based approaches are useful for reasoning about the implementations of fine-grained concurrent data structures whose correctness argument is more sophisticated.

In addition to Sects. 3, 4 and 5, the rest of this paper is organized as follows.

Section 2 We give the technical background necessary for our work.
Section 6 We document ShareInfer [1], a tool that uses the logical tools developed in Sects. 3, 4 and 5 to infer frames and antiframes and check the necessary side conditions. We benchmark ShareInfer with 27 selective examples.
Section 7 We introduce *scaling separation algebra* that allows us to construct predicate multiplication on an abstract structure in a compositional way. We show such model can be constructed from Dockins *et al.*'s tree shares [21]. The key technical proofs in Sects. 5 and 7 have been verified in Coq [1].
Section 8 We prove that there are no useful share models that simultaneously satisfy disjointness and two distributivity axioms. Consequently, at least one axioms has to be removed, which we choose to be the left distributivity. We also prove that the failure of two-sided distributivity forces a side condition on a key proof rule for predicate multiplication.
Section 9 We discuss related work before delivering our conclusion.

$$\begin{array}{ll}
\texttt{root} \overset{0.3}{\mapsto} (3, \texttt{left}, \texttt{right}) & \star \\
\texttt{left} \overset{0.3}{\mapsto} (1, \mathsf{null}, \texttt{grand}) & \star \\
\texttt{right} \overset{0.3}{\mapsto} (4, \texttt{grand}, \mathsf{null}) & \star \\
\texttt{grand} \overset{0.6}{\mapsto} (1, \mathsf{null}, \mathsf{null}) &
\end{array}$$

$$\begin{tikzcd}
 & \boxed{\texttt{root}:3} \arrow[dl, "0.3"'] \arrow[dr, "0.3"] & \\
\boxed{\texttt{left}:1} \arrow[dr, "0.3"'] & & \boxed{\texttt{right}:4} \arrow[dl, "0.3"] \\
 & \boxed{\texttt{grand}:1} &
\end{tikzcd}$$

Fig. 1. This heap satisfies $\mathsf{tree}(\texttt{root}, 0.3)$ despite being a DAG

2 Technical Preliminaries

Share Models. An (additive) share model $(\mathcal{S}, \oplus)$ is a partial commutative monoid with a bottom/empty element $\mathcal{E}$ and top/full element $\mathcal{F}$. On the rationals in $[0, 1]$, $\oplus$ is partial addition, $\mathcal{E}$ is 0, and $\mathcal{F}$ is 1. We also require the existence of complements $\overline{\pi}$ satisfying $\pi \oplus \overline{\pi} = \mathcal{F}$; in $\mathbb{Q}$, $\overline{\pi} \overset{\text{def}}{=} 1 - \pi$.

Separation Logic. Our base separation logic has the following connectives:

$$P, Q, \text{ etc.} \quad \overset{\text{def}}{=} \quad \langle F \rangle \mid P \wedge Q \mid P \vee Q \mid \neg P \mid P \star Q \mid \forall x.P \mid \exists x.P \mid \mu X.P \mid e_1 \overset{\pi}{\mapsto} e_2$$

Pure facts F are put in angle brackets, *e.g.* $\langle even(12) \rangle$. Pure facts force the empty heap, *i.e.* the usual separation logic emp predicate is just a macro for $\langle \top \rangle$. Our propositional fragment has (classical) conjunction $\wedge$, disjunction $\vee$, negation $\neg$, and the separating conjunction $\star$. We have both universal $\forall$ and existential $\exists$ quantifiers, which can be impredicative if desired. To construct recursive predicates we have the usual Tarski least fixpoint μ. The fractional points-to $e_1 \overset{\pi}{\mapsto} e_2$ means we own the π-fraction of the memory cell pointed to by e_1, whose contents is e_2, and nothing more. To distinguish points-to from emp we require that π be non-$\mathcal{E}$. For notational convenience we sometimes elide the full share $\mathcal{F}$ over a fractional maps-to, writing just $e_1 \mapsto e_2$. The connection of $\oplus$ to the fractional maps-to predicate is given by the bi-entailment:

$$\frac{}{e \overset{\pi_1}{\mapsto} e_1 \star e \overset{\pi_2}{\mapsto} e_2 \quad \dashv\vdash \quad e \overset{\pi_1 \oplus \pi_2}{\longmapsto} e_1 \wedge e_1 = e_2} \; \textsc{MapsTo Split}$$

Disjointness. Although intuitive, the rationals are not a good model for shares in SL. Consider this definition for π-fractional trees rooted at x:

$$\mathsf{tree}(x, \pi) \quad \overset{\text{def}}{=} \quad \langle x = \mathsf{null} \rangle \vee \exists d, l, r.\ x \overset{\pi}{\mapsto} (d, l, r) \star \mathsf{tree}(l, \pi) \star \mathsf{tree}(r, \pi) \qquad (1)$$

This tree predicate is obtained directly from the standard recursive predicate for binary trees by asserting only π ownership of the root and recursively doing the same for the left and right substructures, and so at first glance looks straightforward[1]. The problem is that when $\pi \in (0, 0.5]$, then tree can describe some

[1] We write $x \overset{\pi}{\mapsto} (v_1, \ldots, v_n)$ for $x \overset{\pi}{\mapsto} v_1 \star (x + 1) \overset{\pi}{\mapsto} v_2 \star \ldots \star (x + n - 1) \overset{\pi}{\mapsto} v_n$.

non-tree directed acyclic graphs as in Fig. 1. Fractional `trees` are a little too easy to introduce and thus unexpectedly painful to eliminate.

To prevent the deformation of recursive structures shown in Fig. 1, we want to recover the "disjointness" property of basic SL: $e \mapsto e_1 \star e \mapsto e_2 \dashv\vdash \bot$. Disjointness can be specified either as an inference rule in separation logic [41] or as an algebraic rule on the share model [21] as follows:

$$\frac{}{e \overset{\pi}{\mapsto} e_1 \star e \overset{\pi}{\mapsto} e_2 \quad \dashv\vdash \quad \bot}\ {\scriptsize \textsc{MapsTo Disjoint}} \qquad\qquad \forall a, b.\ a \oplus a = b \quad \Rightarrow \quad a = \mathcal{E} \qquad (2)$$

In other words, **a nonempty share π cannot join with itself**. In Sect. 3 we will see how disjointness enables the distribution of predicate multiplication over $\star$ and in Sect. 4 we will see how disjointness enables antiframe inference during biabduction.

Tree Shares. Dockins *et al.* [21] proposed "tree shares" as a share model satisfying disjointness. For this paper the details of the model are not critical so we provide only a brief overview. A tree share $\tau \in \mathbb{T}$ is a binary tree with Boolean leaves, *i.e.* $\tau = \bullet \mid \circ \mid \widehat{\tau_1\ \tau_2}$, where $\circ$ is the empty share $\mathcal{E}$ and $\bullet$ is the full share $\mathcal{F}$. There are two "half" shares: $\widehat{\circ\ \bullet}$ and $\widehat{\bullet\ \circ}$, and four "quarter" shares, *e.g.* $\widehat{\widehat{\bullet\ \circ}\ \circ}$. Trees must be in *canonical form*, *i.e.*, the most compact representation under $\cong$:

$$\frac{}{\circ \cong \circ} \qquad \frac{}{\bullet \cong \bullet} \qquad \frac{}{\circ \cong \widehat{\circ\ \circ}} \qquad \frac{}{\bullet \cong \widehat{\bullet\ \bullet}} \qquad \frac{\tau_1 \cong \tau_1' \quad \tau_2 \cong \tau_2'}{\widehat{\tau_1\ \tau_2} \cong \widehat{\tau_1'\ \tau_2'}}$$

Union $\sqcup$, intersection $\sqcap$, and complement $\bar{\cdot}$ are the basic operations on tree shares; they operate leafwise after unfolding the operands under $\cong$ into the same shape:

$$\widehat{\widehat{\bullet\ \circ}\ \circ} \sqcup \widehat{\widehat{\circ\ \bullet}\ \widehat{\bullet\ \circ}} \quad \cong \quad \widehat{\widehat{\bullet\ \circ}\ \widehat{\circ\ \circ}} \sqcup \widehat{\widehat{\circ\ \bullet}\ \widehat{\bullet\ \circ}} \quad = \quad \widehat{\widehat{\bullet\ \bullet}\ \widehat{\bullet\ \circ}} \quad \cong \quad \widehat{\bullet\ \widehat{\bullet\ \circ}}$$

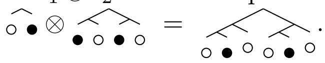
The structure $\langle \mathbb{T}, \sqcup, \sqcap, \bar{\cdot}, \circ, \bullet \rangle$ forms a countable atomless Boolean algebra and thus enjoys decidable existential and first-order theories with precisely known complexity bounds [34]. The join operator $\oplus$ on trees is defined as $\tau_1 \oplus \tau_2 = \tau_3 \overset{\text{def}}{=} \tau_1 \sqcup \tau_2 = \tau_3 \wedge \tau_1 \sqcap \tau_2 = \circ$. Due to their good metatheoretic and computational properties, a variety of program logics [24,25] and verification tools [3,26,33,47] have used tree shares (or other isomorphic structures [19]).

3 Predicate Multiplication

The additive structure of share models is relatively well-understood [21,33,34]. The focus for this paper is exploring the benefits and consequences of incorporating a multiplicative operator $\otimes$ into a share model. The simplest motivation for multiplication is computationally dividing some share π of a resource "in half;"

```
1  struct tree {int d; struct tree* l; struct tree* r;};
2  void processTree(struct tree* x) {
3    if (x == 0) { return; }
                                     ||
4    print(x -> d);                  || 7   print(x -> d);
5    processTree(x -> l);            || 8   processTree(x -> l);
6    processTree(x -> r);            || 9   processTree(x -> r);
                                     ||
10 }
```

Fig. 2. The parallel `processTree` function, written in a C-like language

the two halves of the resource are then given to separate threads for parallel processing. When shares themselves are rationals, $\otimes$ is just ordinary multiplication, *e.g.* we can divide $0.6 = (0.5 \otimes 0.6) \oplus (0.5 \otimes 0.6)$. Defining a notion of multiplication on a share model that satisfies disjointness is somewhat trickier, but we can do so with tree shares $\mathbb{T}$ as follows. Define $\tau_1 \otimes \tau_2$ to be the operation that replaces each $\bullet$ in τ_2 with a copy of τ_1, *e.g.*:  . The structure $(\mathbb{T}, \oplus, \otimes)$ is a kind of "near-semiring." The $\otimes$ operator is associative, has identity $\mathcal{F}$ and null point $\mathcal{E}$, and is right distributive, *i.e.* $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$. It is not commutative, does not distribute on the left, or have inverses. It is hard to do better: adding axioms like multiplicative inverses forces any model satisfying disjointness ($\forall a, b.\ a \oplus a = b \;\Rightarrow\; a = \mathcal{E}$) to have no more than two elements (Sect. 8).

Now consider the toy program in Fig. 2. Starting from the tree rooted at `x`, the program itself is dead simple. First (line 3) we check if the `x` is null, *i.e.* if we have reached a leaf; if so, we `return`. If not, we split into parallel threads (lines 4–6 and 7–9) that do some processing on the root data in both branches. In the toy example, the processing just `print`s out the root data (lines 4 and 7); the `print` command is unimportant: what is important that we somehow access some of the data in the tree. After processing the root, both parallel branches call the `processTree` function recursively on the left `x->l` (lines 5 and 8) and right `x->r` (lines 6 and 9) branches, respectively. After both parallel processes have terminated, the function returns (line 10). The program is simple, so we would like its verification to be equally simple.

Predicate multiplication is the tool that leads to a simple proof. Specifically, we would like to verify that `processTree` has the specification:

$$\forall \pi, x.\ \ \big(\ \{\pi \cdot \mathsf{tree}(x)\}\ \ \texttt{processTree}(x)\ \ \{\pi \cdot \mathsf{tree}(x)\}\ \big)$$

Here $\mathsf{tree}(x) \overset{\text{def}}{=} \langle x = \mathsf{null} \rangle \vee \exists d, l, r.\ x \mapsto (d, l, r) \star \mathsf{tree}(l) \star \mathsf{tree}(r)$ is exactly the usual definition of binary trees in separation logic. Predicate multiplication has allowed us to isolate the fractional ownership from the definition; compare with Eq. (1) above. Our precondition and postcondition both say that x is a pointer to a heap-represented π-owned `tree`. Critically, we want to ensure that our π-share at the end of the program is equal to the π-share at the beginning. This way if our initial caller had full $\mathcal{F}$ ownership before calling `processTree`, he will have full ownership afterwards (allowing him to *e.g.* deallocate the tree).

The intuition behind the proof is simple. First in line 3, we check if `x` is null; if so we are in the base case of the `tree` definition and can simply return. If not we can eliminate the left disjunct and can proceed to split the $\star$-separated bits into disjoint subtrees `l` and `r`, and then dividing the ownership of those bits into two "halves". Let $\mathcal{L} \overset{\text{def}}{=} \widehat{\bullet\,\circ}$ and $\mathcal{R} \overset{\text{def}}{=} \overline{\mathcal{L}} = \widehat{\circ\,\bullet}$. When we start the parallel computation on lines 4 and 7 we want to pass the left branch of the computation the $\mathcal{L} \otimes \pi$-share of the spatial resources, and the right branch of the computation the $\mathcal{R} \otimes \pi$. In both branches we then need to show that we can read from the data cell, which in the simple policy we use for this paper boils down to making sure that the product of two non-$\mathcal{E}$ shares cannot be $\mathcal{E}$. This is a basic property for reasonable share models with multiplication. In the remainder of the parallel code (lines 5–6 and 8–9) we need to make recursive calls, which is done by simply instantiating π with $\mathcal{L} \otimes \pi$ and $\mathcal{R} \otimes \pi$ in the recursive specification (as well as `l` and `r` for x). The later half proof after the parallel call is pleasantly symmetric to the first half in which we fold back the original tree predicate by merging the two halves $\mathcal{L} \otimes \pi$ and $\mathcal{R} \otimes \pi$ back into π. Consequently, we arrive at the postcondition $\pi \cdot \mathsf{tree}(x)$, which is identical to the precondition.

3.1 Proof Rules for Predicate Multiplication

In Fig. 4 we put the formal verification for `processTree`, which follows the informal argument very closely. However, before we go through it, let us consider the reason for this alignment: because the key rules for reasoning about predicate multiplication are bidirectional. These rules are given in Fig. 3. The non-spatial rules are all straightforward and follow the basic pattern that predicate multiplication both pushes into and pulls out of the operators of our logic without meaningful side conditions. The DotPure rule means that predicate multiplication ignores pure facts, too. Complicating the picture slightly, predicate multiplication pushes into implication $\Rightarrow$ but does not pull out of it. Combining DotImpl with DotPure we get a one-way rule for negation: $\pi \cdot (\neg P) \vdash \neg \pi \cdot$. We will explain why we cannot get both directions in Sects. 5.1 and 8.

Most of the spatial rules are also simple. Recall that $\mathsf{emp} \overset{\text{def}}{=} \langle \top \rangle$, so DotPure yields $\pi \cdot \mathsf{emp} \dashv\vdash \mathsf{emp}$. The DotFull rule says that $\mathcal{F}$ is the scalar identity on predicates, just as it is the multiplicative identity on the share model itself. The DotDot rule allows us to "collapse" repeated predicate multiplication using share multiplication; we will shortly see how we use it to verify the recursive calls to `processTree`. Similarly, the DotMapsTo rule shows how predicate multiplication combines with basic maps-to by multiplying the associated shares together. All three rules are bidirectional and require no side conditions.

While the last two rules are both bidirectional, they both have side conditions. The DotPlus rule shows how predicate multiplication distributes over $\oplus$. The $\vdash$ direction does not require a side condition, but the $\dashv$ direction we require that P be *precise* in the usual separation logic sense. Precision will be discussed

$$\frac{P \vdash Q}{\pi \cdot P \vdash \pi \cdot Q}\ \textsc{Dot Pos} \qquad \frac{}{\pi \cdot \langle P \rangle \dashv\vdash \langle P \rangle}\ \textsc{Dot Pure} \qquad \frac{}{\pi \cdot (P \Rightarrow Q) \vdash (\pi \cdot P) \Rightarrow (\pi \cdot Q)}\ \textsc{Dot Impl}$$

$$\frac{}{\pi \cdot (P \wedge Q) \dashv\vdash (\pi \cdot P) \wedge (\pi \cdot Q)}\ \textsc{Dot Conj} \qquad \frac{}{\pi \cdot (P \vee Q) \dashv\vdash (\pi \cdot P) \vee (\pi \cdot Q)}\ \textsc{Dot Disj} \qquad \frac{}{\pi \cdot (\neg P) \vdash \neg \pi \cdot P}\ \textsc{Dot Neg}$$

$$\frac{\tau \neq \emptyset}{\pi \cdot \big(\forall x : \tau.\ P(x)\big) \dashv\vdash \forall x : \tau.\ \pi \cdot P(x)}\ \textsc{Dot Univ} \qquad \frac{}{\pi \cdot \big(\exists x : \tau.\ P(x)\big) \dashv\vdash \exists x : \tau.\ \pi \cdot P(x)}\ \textsc{Dot Exis}$$

$$\frac{}{\mathcal{F} \cdot P \dashv\vdash P}\ \textsc{Dot Full} \qquad \frac{}{\pi_1 \cdot (\pi_2 \cdot P) \dashv\vdash (\pi_1 \otimes \pi_2) \cdot P}\ \textsc{Dot Dot} \qquad \frac{}{\pi \cdot x \mapsto y \dashv\vdash x \overset{\pi}{\mapsto} y}\ \textsc{Dot MapsTo}$$

$$\frac{\mathrm{precise}(P)}{(\pi_1 \oplus \pi_2) \cdot P \dashv\vdash (\pi_1 \cdot P) \star (\pi_2 \cdot P)}\ \textsc{Dot Plus} \qquad \frac{P \vdash \mathsf{uniform}(\pi') \qquad Q \vdash \mathsf{uniform}(\pi')}{\pi \cdot (P \star Q) \dashv\vdash (\pi \cdot P) \star (\pi \cdot Q)}\ \textsc{Dot Star}$$

Fig. 3. Distributivity of the scaling operator over pure and spatial connectives

in Sect. 5.2; for now a simple counterexample shows why it is necessary:

$$\mathcal{L} \cdot (x \mapsto a \vee (x+1) \mapsto b) \star \mathcal{R} \cdot (x \mapsto a \vee (x+1) \mapsto b) \quad \not\vdash \quad \mathcal{F} \cdot (x \mapsto a \vee (x+1) \mapsto b)$$

The premise is also consistent with $x \overset{\mathcal{L}}{\mapsto} a \star (x+1) \overset{\mathcal{R}}{\mapsto} b$.

The DotStar rule shows how predicate multiplication distributes into and out of the separating conjunction $\star$. It is also bidirectional. **Crucially, the $\dashv$ direction fails on non-disjoint share models like** $\mathbb{Q}$, which is the "deeper reason" for the deformation of recursive structures illustrated in Fig. 1. On disjoint share models like $\mathbb{T}$, we get equational reasoning $\dashv\vdash$ subject to the side condition of *uniformity*. Informally, $P \vdash \mathsf{uniform}(\pi')$ asserts that any heap that satisfies P has the permission π' uniformly at each of its defined addresses. In Sect. 8 we explain why we cannot admit this rule without a side condition.

In the meantime, let us argue that most predicates used in practice in separation logic are uniform. First, every SL predicate defined in non-fractional settings, such as $\mathsf{tree}(x)$, is $\mathcal{F}$-uniform. Second, P is a π-uniform predicate if and only if $\pi' \cdot P$ is $(\pi' \otimes \pi)$-uniform. Third, the $\star$-conjunction of two π-uniform predicates is also π-uniform. Since a significant motivation for predicate multiplication is to allow standard SL predicates to be used in fractional settings, these already cover many common cases in practice. It is useful to consider examples of non-uniform predicates for contrast. Here are three (we elide the base cases):

$$\begin{aligned}
\mathsf{slist}(x) &\dashv\vdash \exists d, n. \big((\langle d = 17\rangle \star x \overset{\mathcal{L}}{\mapsto} (d,n)) \vee (\langle d \neq 17\rangle \star x \overset{\mathcal{R}}{\mapsto} (d,n))\big) \star \mathsf{slist}(n)\\
\mathsf{dlist}(x) &\dashv\vdash \exists d, n. x \mapsto d, n \star \mathcal{L} \cdot \mathsf{dlist}(n)\\
\mathsf{dtree}(x) &\dashv\vdash \exists d, l, r. x \mapsto d, l, r \star \mathcal{L} \cdot \mathsf{dtree}(l) \star \mathcal{R} \cdot \mathsf{dtree}(r)
\end{aligned}$$

The $\mathsf{slist}(x)$ predicate owns different amounts of permissions at different memory cells depending on the value of those cells. The $\mathsf{dlist}(x)$ predicate owns decreasing amounts of the list, *e.g.* the first cell is owned more than the second, which is owned more than the third. The $\mathsf{dtree}(x)$ predicate is even stranger, owning different amounts of different branches of the tree, essentially depending on the

```
 1 void processTree(struct tree* x) { // $\{\ \pi \cdot \mathsf{tree}(\mathtt{x})\ \}$
 2 // $\Big\{\pi \cdot \Big(\langle \mathtt{x} = \mathtt{null}\rangle \ \vee\ \big(\exists d,l,r.\ \mathtt{x} \mapsto (d,l,r)\ \star\ \mathsf{tree}(l)\ \star\ \mathsf{tree}(r)\big)\Big)\Big\}$
 3 // $\Big\{\Big\langle \mathtt{x} = \mathtt{null}\Big\rangle \ \vee\ \Big(\exists d,l,r.\ \mathtt{x} \overset{\pi}{\mapsto} (d,l,r)\ \star\ \big(\pi \cdot \mathsf{tree}(l)\big)\ \star\ \big(\pi \cdot \mathsf{tree}(r)\big)\Big)\Big\}$
 4   if (x == null) { // $\{\langle \mathtt{x} = \mathtt{null}\rangle\}$
 5   return;} // $\{\ \pi \cdot \mathsf{tree}(\mathtt{x})\ \}$
 6 // $\Big\{\mathtt{x} \overset{\pi}{\mapsto} (d,l,r)\ \star\ \big(\pi \cdot \mathsf{tree}(l)\big)\ \star\ \big(\pi \cdot \mathsf{tree}(r)\big)\Big\}$
 7 // $\Big\{\mathcal{F} \cdot \Big(\mathtt{x} \overset{\pi}{\mapsto} (d,l,r)\ \star\ \big(\pi \cdot \mathsf{tree}(l)\big)\ \star\ \big(\pi \cdot \mathsf{tree}(r)\big)\Big)\Big\}$
 8 // $\Big\{(\mathcal{L} \oplus \mathcal{R}) \cdot \Big(\mathtt{x} \overset{\pi}{\mapsto} (d,l,r)\ \star\ \big(\pi \cdot \mathsf{tree}(l)\big)\ \star\ \big(\pi \cdot \mathsf{tree}(r)\big)\Big)\Big\}$
 9 // $\left\{\begin{array}{l}\Big(\mathcal{L} \cdot \Big(\mathtt{x} \overset{\pi}{\mapsto} (d,l,r)\ \star\ \big(\pi \cdot \mathsf{tree}(l)\big)\ \star\ \big(\pi \cdot \mathsf{tree}(r)\big)\Big)\Big)\ \star \\ \Big(\mathcal{R} \cdot \Big(\mathtt{x} \overset{\pi}{\mapsto} (d,l,r)\ \star\ \big(\pi \cdot \mathsf{tree}(l)\big)\ \star\ \big(\pi \cdot \mathsf{tree}(r)\big)\Big)\Big)\end{array}\right\}$
10 // $\Big\{\mathcal{L} \cdot \Big(\mathtt{x} \overset{\pi}{\mapsto} (d,l,r)\ \star\ \big(\pi \cdot \mathsf{tree}(l)\big)\ \star\ \big(\pi \cdot \mathsf{tree}(r)\big)\Big)\Big\}$          ||
11 // $\Big\{\mathcal{L} \cdot \mathtt{x} \overset{\pi}{\mapsto} (d,l,r)\ \star\ \mathcal{L} \cdot \pi \cdot \mathsf{tree}(l)\ \star\ \mathcal{L} \cdot \pi \cdot \mathsf{tree}(r)\Big\}$          ||
12 // $\Big\{\mathtt{x} \overset{\mathcal{L} \otimes \pi}{\longmapsto} (d,l,r)\ \star\ \big((\mathcal{L} \otimes \pi) \cdot \mathsf{tree}(l)\big)\ \star\ \big((\mathcal{L} \otimes \pi) \cdot \mathsf{tree}(r)\big)\Big\}$          ||
13   print(x -> d);                                                         || ...
14   processTree(x -> l);  processTree(x -> r);                             ||
15 // $\Big\{\mathtt{x} \overset{\mathcal{L} \otimes \pi}{\longmapsto} (d,l,r)\ \star\ \big((\mathcal{L} \otimes \pi) \cdot \mathsf{tree}(l)\big)\ \star\ \big((\mathcal{L} \otimes \pi) \cdot \mathsf{tree}(r)\big)\Big\}$          ||
16 // $\{\mathcal{L} \cdot \pi \cdot \mathtt{x} \mapsto (d,l,r)\ \star\ \mathcal{L} \cdot \pi \cdot \mathsf{tree}(l)\ \star\ \mathcal{L} \cdot \pi \cdot \mathsf{tree}(r)\}$          ||
17 // $\{\mathcal{L} \cdot \pi \cdot \big(\mathtt{x} \mapsto (d,l,r)\ \star\ \mathsf{tree}(l)\ \star\ \mathsf{tree}(r)\big)\}$          ||
18 // $\left\{\begin{array}{l}\Big(\mathcal{L} \cdot \pi \cdot \big(\mathtt{x} \mapsto (d,l,r)\ \star\ \mathsf{tree}(l)\ \star\ \mathsf{tree}(r)\big)\Big)\ \star \\ \Big(\mathcal{R} \cdot \pi \cdot \big(\mathtt{x} \mapsto (d,l,r)\ \star\ \mathsf{tree}(l)\ \star\ \mathsf{tree}(r)\big)\Big)\end{array}\right\}$
19 // $\Big\{(\mathcal{L} \oplus \mathcal{R}) \cdot \pi \cdot \Big(\mathtt{x} \mapsto (d,l,r)\ \star\ \mathsf{tree}(l)\ \star\ \mathsf{tree}(r)\Big)\Big\}$
20 } // $\{\ \pi \cdot \mathsf{tree}(\mathtt{x})\ \}$
```

Fig. 4. Reasoning with the scaling operator $\pi \cdot P$.

path to the root. None of these predicates mix well with DotStar, but perhaps they are not useful to verify many programs in practice, either. In Sects. 5.1 and 5.2 we will discuss how to prove predicates are precise and uniform. In Sect. 5.4 will demonstrate our techniques to do so by applying them to two examples.

3.2 Verification of `processTree` using predicate multiplication

We now explain how the proof of `processTree` is carried out in Fig. 4 using scaling rules in Fig. 3. In line 2, we unfold the definition of predicate $\mathsf{tree}(x)$ which consists of one base case and one inductive case. We reach line 3 by pushing π inward using various rules DotPure, DotDisj, DotExis, DotMapsto and DotStar. To use DotStar we must prove that $\mathsf{tree}(x)$ is $\mathcal{F}$-uniform, which we show how to do in Sect. 5.4. We prove this lemma once and use it many times.

The base base $\texttt{x} = \texttt{null}$ is handled in lines 4–5 by applying rule DotPure, *i.e.*, $\langle \texttt{x} = \texttt{null} \rangle \vdash \pi \cdot \langle \texttt{x} = \texttt{null} \rangle$ and then DotPos, $\pi \cdot \langle \texttt{x} = \texttt{null} \rangle \vdash \pi \cdot \mathsf{tree}(x)$. For the inductive case, we first apply DotFull in line 7 and then replace $\mathcal{F}$ with $\mathcal{L} \oplus \mathcal{R}$ (recall that $\mathcal{R}$ is $\mathcal{L}$'s compliment). On line 9 we use DotPlus to translate the split on shares with $\oplus$ into a split on heaps with $\star$.

We show only one parallel process; the other is a mirror image. Line 10 gives the precondition from the Parallel rule, and then in lines 11 and 12 we continue to "push in" the predicate multiplication. To verify the code in lines 13–14 just requires Frame. Notice that we need the DotDot rule to "collapse" the two uses of predicate multiplication into one so that we can apply the recursive specification (with the new π' in the recursive precondition equal to $\mathcal{L} \otimes \pi$).

Having taken the predicate completely apart, it is now necessary to put Humpty Dumpty back together again. Here is why it is vital that all of our proof rules are bidirectional, without which we would not be able to reach the final postcondition $\pi \cdot \mathsf{tree}(x)$. The final wrinkle is that for line 19 we must prove the precision of the $\mathsf{tree}(x)$ predicate. We show how to do so with example in Sect. 5.4, but typically in a verification this is proved once per predicate as a lemma.

4 Bi-abductive Inference with Fractional Permissions

Biabduction is a separation logic inference process that helps to increase the scalability of verification for sizable programs [22,49]; in recent years it has been the focus of substantial research for (sequential) separation logic [8,10,11,32]. Biabduction aims to infer the missing information in an incomplete separation logic entailment. More precisely, given an incomplete entailment $A \star [??] \vdash B \star [??]$, we would like to find predicates for the two missing pieces [??] that complete the entailment in a nontrivial manner. The first piece is called the *antiframe* while the second is the *inference frame*. The standard approach consists of two sequential subroutines, namely the *abductive inference* and *frame inference* to construct the antiframe and frame respectively. Our task in this section is to show how to upgrade these routines to handle fractional permissions so that biabduction can extend to concurrent programs. As we will see, disjointness plays a crucial role in antiframe inference.

4.1 Fractional Residue Computation

Consider the fractional point-to bi-abduction problem with rationals:

$$a \xmapsto{\pi_1} b \star [??] \vdash a \xmapsto{\pi_2} b \star [??]$$

There are three cases to consider, namely $\pi_1 = \pi_2$, $\pi_1 < \pi_2$ or $\pi_1 > \pi_2$. In the first case, both the (minimal) antiframe F_a and frame F_f are emp; for the second case we have $F_\mathrm{a} = \mathsf{emp}$, $F_\mathrm{f} = a \xmapsto{\pi_2 - \pi_1} b$ and the last case gives us $F_\mathrm{a} = a \xmapsto{\pi_1 - \pi_2} b, F_\mathrm{f} = \mathsf{emp}$. Here we straightforwardly compute the residue permission using rational subtraction. In general, one can attempt to define subtraction $\ominus$ from a share model $\langle \mathcal{S}, \oplus \rangle$ as $a \ominus b = c \overset{\text{def}}{=} b \oplus c = a$. However, this definition is too coarse as we want subtraction to be a total function so that the residue is always computable efficiently. A solution to this issue is to relax the requirements for $\ominus$, asking only that it satisfies the following two properties:

$$C_1 : a \oplus (b \ominus a) = b \oplus (a \ominus b) \qquad C_2 : a \ll b \oplus c \Rightarrow a \ominus b \ll c$$

where $a \ll b \overset{\text{def}}{=} \exists c.\ a \oplus c = b$. The condition C_1 provides a convenient way to compute the fractional residue in both the frame and antiframe while C_2 asserts that $a \ominus b$ is effectively the minimal element that when joined with b becomes greater than a. In the rationals $\mathbb{Q}$, $a \ominus b \overset{\text{def}}{=} \mathit{if}(a > b)\ \mathit{then}\ a - b\ \mathit{else}\ 0$. On tree shares $\mathbb{T}$, $a \ominus b \overset{\text{def}}{=} a \sqcap \bar{b}$. Recalling that the case when $\pi_1 = \pi_2$ is simple (both the antiframe and frame are just emp), then if $\pi_1 \neq \pi_2$ we can compute the fractional antiframe and inference frames uniquely using $\ominus$:

$$\frac{}{a \xmapsto{\pi_1} b \star a \xmapsto{\pi_2 \ominus \pi_1} b \vdash a \xmapsto{\pi_2} b \star a \xmapsto{\pi_1 \ominus \pi_2} b}\ \textsc{Msub}$$

Generally, the following rule helps compute the residue of predicate P:

$$\frac{\text{precise}(P)}{\pi_1 \cdot P \star (\pi_2 \ominus \pi_1) \cdot P \vdash \pi_2 \cdot P \star (\pi_1 \ominus \pi_2) \cdot P}\ \textsc{Psub}$$

Using C_1 and C_2 it is easy to prove that the residue is minimal w.r.t. $\ll$, *i.e.*:

$$\pi_1 \oplus a = \pi_2 \oplus b \Rightarrow \pi_2 \ominus \pi_1 \ll a \wedge \pi_1 \ominus \pi_2 \ll b$$

4.2 Extension of Predicate Axioms

To support reasoning over recursive data structure such as lists or trees, the assertion language is enriched with the corresponding inductive predicates. To derive properties over inductive predicates, verification tools often contain a list of predicate axioms/facts and use them to aid the verification process [9,32]. These facts are represented as entailment rules $A \vdash B$ that can be classified into "folding" and "unfolding" rules to manipulate the representation of inductive predicates. For example, some axioms for the tree predicate are:

$$F_1 : x = 0 \wedge \mathsf{emp} \vdash \mathsf{tree}(x) \qquad F_2 : x \mapsto (v, x_1, x_2) \star \mathsf{tree}(x_1) \star \mathsf{tree}(x_2) \vdash \mathsf{tree}(x)$$
$$U : \mathsf{tree}(x) \wedge x \neq 0 \vdash \exists v, x_1, x_2.\ x \mapsto (v, x_1, x_2) \star \mathsf{tree}(x_1) \star \mathsf{tree}(x_2)$$

We want to transform these axioms into fractional forms. The key ingredient is the DotPos rule from Fig. 3, that lifts the fractional portion of an entailment, *i.e.* $(P \vdash Q) \Rightarrow (\pi \cdot P \vdash \pi \cdot Q)$. Using this and the other scaling rules from Fig. 3, we can upgrade the folding/unfolding rules into corresponding fractional forms:

$$F_1' : x = 0 \wedge \mathsf{emp} \vdash \pi \cdot \mathsf{tree}(x) \qquad F_2' : x \xmapsto{\pi} (v, x_1, x_2) \star \pi \cdot \mathsf{tree}(x_1) \star \pi \cdot \mathsf{tree}(x_2) \vdash \pi \cdot \mathsf{tree}(x)$$
$$U : \mathsf{tree}(x) \wedge x \neq 0 \vdash \exists v, x_1, x_2.\ x \mapsto (v, x_1, x_2) \star \mathsf{tree}(x_1) \star \mathsf{tree}(x_2)$$

As our scaling rules are bi-directional, they can be applied both in the antecedent and consequent to produce a smooth transformation to fractional axioms. Also, recall that our DotStar rule $\pi \cdot (P \star Q) \dashv\vdash \pi \cdot P \star \pi \cdot Q$ has a side condition that both P and Q are π'-uniform. This condition is trivial in the transformation as standard predicates (*i.e.* those without permissions) are automatically $\mathcal{F}$-uniform. Furthermore, the precision and uniformity properties can be transferred directly to fractional forms by the following rules:

$$\text{precise}(\pi \cdot P) \Leftrightarrow \text{precise(P)} \qquad P \vdash \mathsf{uniform}(\pi) \Leftrightarrow \pi' \cdot P \vdash \mathsf{uniform}(\pi' \otimes \pi)$$

4.3 Abductive Inference and Frame Inference

To construct the antiframe, Calcagno *et al.* [10] presented a general framework for antiframe inference which contains rules of the form:

$$\frac{\Delta' \star [M'] \rhd H' \qquad \text{Cond}}{\Delta \star [M] \rhd H}$$

where Cond is the side condition, together with consequents (H, H'), heap formulas (Δ, Δ') and antiframes (M, M'). In principle, the abduction algorithm gradually matches fragments of consequent with antecedent, derives sound equalities among variables while applying various folding and unfolding rules for recursive predicates in both sides of the entailment. Ideally, the remaining unmatched fragments of the antecedent are returned to form the antiframe. During the process, certain conditions need to be maintained, *e.g.*, satisfiability of the antecedent or minimal choice for antiframe. After finding the antiframe, the inference process is invoked to construct the inference frame. In principle, the old antecedent is first combined with the antiframe to form a new antecedent whose fragments are matched with the consequent. Eventually, the remaining unmatched fragments of the antecedent are returned to construct the inference frame.

The discussion of fractional residue computation in Sect. 4.1 and extension of recursive predicate rules in Sect. 4.2 ensure a smooth upgrade of the biabduction algorithm to fractional form. We demonstrate this intuition using the example in Fig. 5. The partial consequent is a fractional $\mathsf{tree}(x)$ predicate with permission π_3 while the partial antecedent is star conjunction of a fractional maps-to predicate of address x with permission π_1, a fractional $\mathsf{tree}(x_1)$ predicate with permission π_2 and a null pointer x_2. Following the spirit of Calcagno *et al.* [10], the steps in both sub-routines include applying the folding and unfolding rules for predicate tree and then matching the corresponding pair of fragments from antecedent and consequent. On the other hand, the upgraded part is reflected through the use of the two new rules Msub and Psub to compute the fractional residues as well as a more general system of folding and unfolding rules for predicate tree. We are then able to compute the antiframe $a = x_1 \wedge (\pi_3 \ominus \pi_2) \cdot \mathsf{tree}(x_1) \star x \xmapsto{\pi_3 \ominus \pi_2} (v, a, x_2)$ and the inference frame $x \xmapsto{\pi_1 \ominus \pi_3} (v, x_1, x_2) \star (\pi_2 \ominus \pi_3) \cdot \mathsf{tree}(x_1)$ respectively.

$$x \xmapsto{\pi_1} (v,a,x_2) \star \pi_2 \cdot \mathsf{tree}(x_1) \star (x_2 = 0 \wedge \mathsf{emp}) \star [??] \;\vdash\; \pi_3 \cdot \mathsf{tree}(x) \star [??]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(x_2 = 0 \wedge \mathsf{emp}) \star [\mathsf{emp}] \rhd \mathsf{emp}}\;\textsc{Base}}{(x_2 = 0 \wedge \mathsf{emp}) \star [\mathsf{emp}] \rhd \pi_3 \cdot \mathsf{tree}(x_2)}\;\mathsf{F_1}'}{\pi_2 \cdot \mathsf{tree}(x_1) \star (x_2 = 0 \wedge \mathsf{emp}) \star [(\pi_3 \ominus \pi_2) \cdot \mathsf{tree}(x_1)] \rhd \pi_3 \cdot \mathsf{tree}(x_1) \star \pi_3 \cdot \mathsf{tree}(x_2)}\;\begin{matrix}\textsc{Psub}\\ \textsc{Msub}\end{matrix}}{\dfrac{\begin{matrix} x \xmapsto{\pi_1} (v,a,x_2) \star \pi_2 \cdot \mathsf{tree}(x_1) \star (x_2 = 0 \wedge \mathsf{emp}) \star [(\pi_3 \ominus \pi_2) \cdot \mathsf{tree}(x_1) \\ \star x \xmapsto{\pi_3 \ominus \pi_1} (v,a,x_2)] \rhd x \xmapsto{\pi_3} (v,a,x_2) \star \pi_3 \cdot \mathsf{tree}(x_1) \star \pi_3 \cdot \mathsf{tree}(x_2)\end{matrix}}{\begin{matrix} x \xmapsto{\pi_1} (v,a,x_2) \star \pi_2 \cdot \mathsf{tree}(x_1) \star (x_2 = 0 \wedge \mathsf{emp}) \\ \star\; [a = x_1 \wedge (\pi_3 \ominus \pi_2) \cdot \mathsf{tree}(x_1) \star x \xmapsto{\pi_3 \ominus \pi_1} (v,a,x_2)] \rhd \pi_3 \cdot \mathsf{tree}(x)\end{matrix}}\;\begin{matrix}\textsc{match}\\ +\mathrm{F}_2'\end{matrix}}$$

Abductive inference

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{emp} \rhd \mathsf{emp} \star [\mathsf{emp}]}\;\textsc{Base}}{x \xmapsto{\pi_1 \oplus (\pi_3 \ominus \pi_1)} (v,x_1,x_2) \rhd x \xmapsto{\pi_3} (v,x_1,x_2) \star [x \xmapsto{(\pi_1 \ominus \pi_3)} (v,x_1,x_2)]}\;\textsc{Msub}}{\begin{matrix} x \xmapsto{\pi_1 \oplus (\pi_3 \ominus \pi_1)} (v,x_1,x_2) \star (\pi_2 \oplus (\pi_3 \ominus \pi_2)) \cdot \mathsf{tree}(x_1) \rhd \\ x \xmapsto{\pi_3} (v,x_1,x_2) \star \pi_3 \cdot \mathsf{tree}(x_1) \star [x \xmapsto{(\pi_1 \ominus \pi_3)} (v,x_1,x_2) \star (\pi_2 \ominus \pi_3) \cdot \mathsf{tree}(x_1)]\end{matrix}}\;\textsc{Psub}}{\begin{matrix} x \xmapsto{\pi_1 \oplus (\pi_3 \ominus \pi_1)} (v,x_1,x_2) \star (\pi_2 \oplus (\pi_3 \ominus \pi_2)) \cdot \mathsf{tree}(x_1) \star (x_2 = 0 \wedge \mathsf{emp}) \rhd \\ x \xmapsto{\pi_3} (v,x_1,x_2) \star \pi_3 \cdot \mathsf{tree}(x_1) \star \pi_3 \cdot \mathsf{tree}(x_2) \star [x \xmapsto{(\pi_1 \ominus \pi_3)} (v,x_1,x_2) \star (\pi_2 \ominus \pi_3) \cdot \mathsf{tree}(x_1)]\end{matrix}}\;\mathrm{F}_1'$$

Frame inference

Fig. 5. An example of biabduction with fractional permissions

Antiframe Inference and Disjointness. Consider the following abduction problem:

$$x \mapsto (v,x_1,x_2) \star \mathsf{tree}(x_1) \star [??] \vdash \mathsf{tree}(x)$$

Using the folding rule F_2, we can identify the antiframe as $\mathsf{tree}(x_2)$. Now suppose we have a rational permission $\pi \in \mathbb{Q}$ distributed everywhere, *i.e.*:

$$x \xmapsto{\pi} (v,x_1,x_2) \star \pi \cdot \mathsf{tree}(x_1) \star [??] \vdash \pi \cdot \mathsf{tree}(x)$$

A naïve solution is to let the antiframe be $\pi \cdot \mathsf{tree}(x_2)$. However, in $\mathbb{Q}$ this choice is unsound due to the deformation of recursive structures issue illustrated in Fig. 1: if the antiframe is $\pi \cdot \mathsf{tree}(x_2)$, the left hand side can be a DAG, even though the right hand side must be a tree. However, in disjoint share models like $\mathbb{T}$, choosing $\pi \cdot \mathsf{tree}(x_2)$ for the antiframe is correct and the entailment holds. As is often the case, things are straightforward once the definitions are correct.

5 A Proof Theory for Fractional Permissions

Our main objective in this section is to show how to discharge the uniformity and precision side conditions required by the DotStar and DotPlus rules.

To handle recursive predicates like $\mathsf{tree}(x)$ we develop set of novel modal-logic based proof rules to carry out induction in the heap. To allow tools to leverage existing entailment checkers, all of these techniques are done **in the object logic itself**, rather than in the metalogic. Thus, in Sect. 5, we do not assume a concrete model for our object logic (in Sect. 7 we will develop a model).

First we discuss new proof rules for predicate multiplication and fractional maps-to (Sect. 5.1), precision (Sect. 5.2), and induction over fractional heaps (Sect. 5.3). We then conclude (Sect. 5.4) with two examples of proving real properties using our proof theory: that $\mathsf{tree}(x)$ is $\mathcal{F}$-uniform and that $\mathsf{list}(x)$ is precise. Some of the theorems have delicate proofs, so all of them have been verified in Coq [1].

5.1 Proof Theory for Predicate Multiplication and Fractional Maps-To

In Sect. 3 we presented the key rules that someone who wants to verify programs using predicate multiplication is likely to find convenient. On page 13 we present a series of additional rules, mostly used to establish the "uniform" and "precise" side conditions necessary in our proofs.

Figure 6 is the simplest group, giving basic facts about the fractional points-to predicate. Only $\mapsto$ INVERSION is not immediate from the nonfractional case. It says that it is impossible to have two fractional maps-tos of the same address and with two different values. We need this fact to *e.g.* prove that predicates with existentials such as tree are precise.

$$\dfrac{}{(x \overset{\pi}{\mapsto} y_1 \star \top) \wedge (x \overset{\pi'}{\mapsto} y_2 \star \top) \vdash \lfloor y_1 = y_2 \rfloor}\ {\scriptstyle \mapsto\text{INVERSION}} \qquad \dfrac{}{x \overset{\pi}{\mapsto} y \vdash \neg\mathsf{emp}}\ {\scriptstyle \mapsto\mathsf{emp}} \qquad \dfrac{}{x \overset{\pi}{\mapsto} y \vdash \lfloor x \neq \mathtt{null} \rfloor}\ {\scriptstyle \mapsto\mathtt{null}}$$

Fig. 6. Proof theory for fractional maps-to

$$\dfrac{}{\mathsf{emp} \vdash \mathsf{uniform}(\pi)}\ {\scriptstyle \mathsf{uniform/emp}} \qquad \dfrac{}{\mathsf{uniform}(\pi) \star \mathsf{uniform}(\pi) \dashv\vdash \mathsf{uniform}(\pi)}\ {\scriptstyle \mathsf{uniform}\star}$$

$$\dfrac{P \vdash \mathsf{uniform}(\pi)}{\pi' \cdot P \vdash \mathsf{uniform}(\pi' \otimes \pi)}\ {\scriptstyle \mathsf{uniformDot}} \qquad \dfrac{}{\mathrm{precise}(x \overset{\pi}{\mapsto} y)}\ {\scriptstyle \mapsto\text{PRECISE}}$$

$$\dfrac{}{x \overset{\pi}{\mapsto} y \vdash \mathsf{uniform}(\pi)}\ {\scriptstyle \mapsto\mathsf{uniform}} \qquad \dfrac{\mathrm{precise}(P)}{\mathrm{precise}(\pi \cdot P)}\ {\scriptstyle \text{DOT PRECISE}}$$

Fig. 7. Uniformity and precision for predicate multiplication

$$\frac{G \vdash \mathsf{precisely}(P) \quad G \vdash \mathsf{precisely}(Q)}{G \vdash \mathsf{precisely}(P \star Q)}\ \mathsf{precisely}\star \qquad \frac{\top \vdash \mathsf{precisely}(P)}{\mathrm{precise}(P)}\ \begin{matrix}\mathsf{precisely}\\ \textsc{Precise}\end{matrix}$$

$$\frac{}{\mathsf{precisely}(P) \vdash \big((P \star Q) \wedge (P \star R)\big) \Rightarrow \big(P \star (Q \wedge R)\big)}\ \begin{matrix}\mathsf{precisely}\\ \textsc{Left}\end{matrix} \qquad \frac{\exists x.\Big(G \vdash \mathsf{precisely}\big(P(x)\big)\Big)}{G \vdash \mathsf{precisely}\big(\forall x. P(x)\big)}\ \mathsf{precisely}\forall$$

$$\frac{\forall Q, R.\Big(G \vdash \big((P \star Q) \wedge (P \star R)\big) \Rightarrow \big(P \star (Q \wedge R)\big)\Big)}{G \vdash \mathsf{precisely}(P)}\ \begin{matrix}\mathsf{precisely}\\ \textsc{Right}\end{matrix} \qquad \frac{G \vdash \mathsf{precisely}(P)}{G \vdash \mathsf{precisely}(P \wedge Q)}\ \mathsf{precisely}\wedge$$

$$\frac{\begin{matrix}\forall x.\Big(G \vdash \mathsf{precisely}\big(P(x)\big)\Big) \\ \forall x, y.\Big(G \wedge \big(P(x) \star \top\big) \wedge \big(P(y) \star \top\big) \vdash \lvert x = y \rvert\Big)\end{matrix}}{G \vdash \mathsf{precisely}\big(\exists x. P(x)\big)}\ \mathsf{precisely}\exists \qquad \frac{\begin{matrix} G \vdash \mathsf{precisely}(P) \\ G \vdash \mathsf{precisely}(Q) \\ G \wedge (P \star \top) \wedge (Q \star \top) \vdash \bot \end{matrix}}{G \vdash \mathsf{precisely}(P \vee Q)}\ \mathsf{precisely}\vee$$

Fig. 8. Proof theory for precision

$$\frac{}{\circledcirc P \vdash P}\ \mathrm{T} \qquad \frac{}{\circledcirc P \vdash \circledcirc\circledcirc P}\ \circledcirc\circledcirc \qquad \frac{}{\rhd_\pi P \vdash \rhd_\pi \rhd_\pi P}\ \rhd_\pi \rhd_\pi$$

$$\frac{\rhd_\pi P \vdash P}{\top \vdash P}\ \mathrm{W} \qquad \frac{}{\rhd_\pi P \dashv\vdash \rhd_\pi \circledcirc P}\ \rhd_\pi \circledcirc \qquad \frac{}{\rhd_\pi P \dashv\vdash \circledcirc \rhd_\pi P}\ \circledcirc \rhd_\pi$$

$$\frac{}{(P \star Q) \wedge \circledcirc R \vdash (P \wedge \circledcirc R) \star (Q \wedge \circledcirc R)}\ \circledcirc\star \qquad \frac{P \vdash U(\pi) \wedge \neg\mathsf{emp}}{(P \star Q) \wedge \rhd_\pi R \vdash (P \wedge \rhd_\pi R) \star (Q \wedge R)}\ \rhd_\pi\star$$

Fig. 9. Proof theory for substructural induction

Proving the side conditions for DotPlus *and* DotStar. Figure 7 contains some rules for establishing that P is π-uniform (*i.e.* $P \vdash \mathsf{uniform}(\pi)$) and that P is precise. Since uniformity is a simple property, the rules are easy to state:

To use predicate multiplication we will need to prove two kinds of side conditions: $\mathsf{uniform/emp}$ tells us that emp is π-uniform for all π; the conclusion (all defined heap locations are held with share π) is vacuously true. The $\mathsf{uniform}$Dot rule tells us that if P is π-uniform then when we multiply P by a fraction π' the result is $(\pi' \otimes \pi)$-uniform. The $\mapsto$ $\mathsf{uniform}$ rule tells us that points-to is uniform. The $\mathsf{uniform}\star$ rule possesses interesting characteristics. The $\dashv$ direction follows from $\mathsf{uniform/emp}$ and the $\star\mathsf{emp}$ rule ($P \star \mathsf{emp} \dashv\vdash P$). The $\vdash$ direction is not automatic but very useful. One consequence is that from $P \vdash \mathsf{uniform}(\pi)$ and $Q \vdash \mathsf{uniform}(\pi)$ we can prove $P \star Q \vdash \mathsf{uniform}(\pi)$. The $\vdash$ direction follows from disjointness but fails over non-disjoint models such as rationals $\mathbb{Q}$.

The $\mapsto$ precise rule tells us that points-tos are precise. The DotPrecise rule is a partial solution to proving precision. It states that $\pi \cdot P$ is precise if and only if P is precise. We will next show how to prove that P itself is precise.

5.2 Proof Theory for Proving that Predicates Are Precise

Proving that a predicate is π-uniform is relatively straightforward using the proof rules presented so far. However, proving that a predicate is precise is not as pleasant. Traditionally precision is defined (and checked for concrete predicates) in the metalogic [40] using the following definition:

$$\text{precise}(P) \stackrel{\text{def}}{=} \forall h, h_1, h_2.\ h_1 \subseteq h \Rightarrow h_2 \subseteq h \Rightarrow (h_1 \models P) \Rightarrow (h_2 \models P) \Rightarrow h_1 = h_2 \quad (3)$$

Here we write $h_1 \subseteq h_2$ to mean that h_1 is a subheap of h_2, *i.e.* $\exists h'.h_1 \oplus h' = h_2$, where $\oplus$ is the joining operation on the underlying separation algebra [21]. Essentially precision is a kind of uniqueness property: if a predicate P is precise then it can only be true on a single subheap.

Rather than checking precision in the metalogic, we wish to do so in the object logic. We give a proof theory that lets us do so in Fig. 8. Among other advantages, proving precision in the object logic lets tools build on existing separation logic entailment checkers to prove the precision of recursive predicates. The core idea is simple: we define a new object logic operator "$\mathsf{precisely}(P)$" that captures the notion of precision relativized to the current heap; essentially it is a partially applied version of the definition of precise(P) in Eq. (3):

$$h \models \mathsf{precisely}(P) \stackrel{\text{def}}{=} \forall h_1, h_2.h_1 \subseteq h \Rightarrow h_2 \subseteq h \Rightarrow (h_1 \models P) \Rightarrow (h_2 \models P) \Rightarrow h_1 = h_2 \quad (4)$$

Although we have given $\mathsf{precisely}$'s model to aid intuition, we emphasize that in Sect. 5 all of our proofs take place in the object logic; we never unfold $\mathsf{precisely}$'s definition. Note that $\mathsf{precisely}$ is also generally weaker than the typical notion of precision. For example, the predicate $x \mapsto 7 \vee y \mapsto 7$ is not precise; however the entailment $z \mapsto 8 \vdash \mathsf{precisely}(x \mapsto 7 \vee y \mapsto 7)$ is provable from Fig. 8.

That said, two notions are closely connected as given in the $\mathsf{precisely}$PRECISE rule. We also give introduction $\mathsf{precisely}$RIGHT and elimination rules $\mathsf{precisely}$LEFT that make a connection between precision and an "antidistribution" of $\star$ over $\wedge$.

We also give a number of rules for showing how $\mathsf{precisely}$ combines with the connectives of our logic. The rules for propositional $\wedge$ and separating $\star$ conjunction follow well-understood patterns, with the addition of an arbitrary premise context G being the key feature. The rule for disjunction $\vee$ is a little trickier, with an additional premise that forces the disjunction to be exclusive rather than inclusive. An example of such an exclusive disjunction is in the standard definition of the tree predicate, where the first disjunct $\langle x = \texttt{null}\rangle$ is fundamentally incompatible with the second disjunct $\exists d, l, r.x \mapsto d, l, r \star \ldots$ since $\mapsto$ does not allow the address to be $\texttt{null}$ (by rule $\mapsto \texttt{null}$ from Fig. 6). The rules for universal quantification $\forall$ existential quantification $\exists$ are essentially generalizations of the rules for the traditional conjunction $\wedge$ and disjunction $\vee$.

It is now straightforward to prove the precision of simple predicates such as $\langle x = \texttt{null}\rangle \vee (\exists y.x \mapsto y \star y \mapsto 0)$. Finding and proving the key lemmas that enable the proof of the precision of recursive predicates remains a little subtle.

5.3 Proof Theory for Induction over the Finiteness of the Heap

Recursive predicates such as $\mathsf{list}(x)$ and $\mathsf{tree}(x)$ are common in SL. However, proving properties of such predicates, such as proving that $\mathsf{list}(x)$ is precise, is a little tricky since the μFoldUnfold rule provided by the Tarski fixed point does not automatically provide an induction principle. Generally speaking such properties follow by some kind of induction argument, either over auxiliary parameters (*e.g.* if we augment trees to have the form $\mathsf{tree}(x, \tau)$, where τ is an inductively-defined type in the metalogic) or over the finiteness of the heap itself. Both arguments usually occur in the metalogic rather than the object logic.

We have two contributions to make for proving inductive properties. First, we show how to do induction over the heap in a fractional setting. Intuitively this is more complicated than in the non-fractional case because there are infinite sequences of strictly smaller subheaps. That is, for a given initial heap h_0, there are infinite sequences h_1, h_2, ...such that $h_0 \supsetneq h_1 \supsetneq h_2 \supsetneq \ldots$. The disjointness property does not fundamentally change this issue, so we illustrate with an example with the shares in $\mathbb{Q}$. The heap h_0 satisfying $x \overset{1}{\mapsto} y$ is strictly larger than the heap h_1 satisfying $x \overset{\frac{1}{2}}{\mapsto} y$, which is strictly larger than the heap h_2 satisfying $x \overset{\frac{1}{4}}{\mapsto} y$; in general h_i satisfies $x \overset{\frac{1}{2^i}}{\mapsto} y$. Since our sequence is infinite, we cannot use it as the basis for an induction argument. The solution is that we require that the heaps decrease by at least some constant size c. If each heap subsequent heap must shrink by at least *e.g.* $c = 0.25$ of a memory cell then the sequence must be finite just as in the non-fractional case, *i.e.* $c = \mathcal{F}$. More sophisticated approaches are conceivable (*e.g.* limits) but they are not easy to automate and we did not find any practical examples that require such methods.

Our second contribution is the development of a proof theory in the object logic that can carry out these kinds of induction proofs in a relatively straightforward way. The proof rules that let us do so are given in Fig. 9. Once good lemmas are identified, we find doing induction proofs over the finite heap formally in the object logic simpler than doing the same proofs in the metalogic.

The key to our induction rules is two new operators: "within" $\circledcirc$ and "shrinking" $\rhd_\pi$. Essentially $\rhd_\pi P$ is used as an induction guard, preventing us from applying our induction hypothesis P until we are on a π-smaller subheap. When $\pi = \mathcal{F}$ we sometimes write just $\rhd P$. Semantically, if h satisfies $\rhd_\pi P$ then P is true **on all strict subheaps of h that are smaller by at least a π-piece**. Accordingly, the key elimination rule $\rhd_\pi\star$ may seem natural: it verifies that the induction guard is satisfied and unlocks the underlying hypothesis. To start an induction proof to prove an arbitrary goal $\top \models P$, we use the rule W to introduce an induction hypothesis, resulting in the new entailment goal of $\rhd_\pi P \vdash P$.

Some definitions, such as $\mathsf{list}(x)$, have only one "recursive call"; others, such as $\mathsf{tree}(x)$ have more than one. Moreover, sometimes we wish to apply our inductive hypothesis immediately after satisfying the guard, whereas other times it is convenient to satisfy the guard somewhat before we need the inductive hypothesis. To handle both of these issues we use the "within" operator $\circledcirc$ such that $h \models \circledcirc P$ means P is true on all subheaps of h, which is the intuition behind the rule $\circledcirc\star$. To apply our induction hypothesis somewhat after meeting its guard (or if we wish to apply it more than once) we use the $\triangleright_\pi\circledcirc$ rule to add the $\circledcirc$ modality before eliminating the guard. We will see an example of this shortly.

5.4 Using Our Proof Theory

We now turn to two examples of using our proof theory from page 13 to demonstrate that the rule set is strong and flexible enough to prove real properties.

Proving that $\mathsf{tree}(x)$ *is* $\mathcal{F}$*-uniform.* Our logical rules for induction and uniformity are able to establish the uniformity of predicates in a fairly simple way. Here we focus on the $\mathsf{tree}(x)$ predicate because it is a little harder due to the two recursive "calls" in its unfolding. For convenience, we will write $\mathsf{u}(\pi)$ instead of $\mathsf{uniform}(\pi)$.

Our initial proof goal is $\mathsf{tree}(x) \vdash \mathsf{u}(\mathcal{F})$. Standard natural deduction arguments then reach the goal $\top \vdash \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{u}(\mathcal{F})$, after which we apply the W rule ($\pi = \mathcal{F}$ is convenient) to start the induction, adding the hypothesis $\triangleright\forall x.\mathsf{tree}(x) \Rightarrow \mathsf{u}(\mathcal{F})$, which we strengthen with the $\triangleright_\pi\circledcirc$ rule to reach $\triangleright \circledcirc \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{u}(\mathcal{F})$. Natural deduction from there reaches

$$\big(\langle x = \texttt{null}\rangle \vee \exists d, l, r. x \mapsto (d, l, r) \star \mathsf{tree}(l) \star \mathsf{tree}(r)\big) \wedge \big(\triangleright \circledcirc \forall x.\mathsf{tree}(x) \Rightarrow \mathsf{u}(\mathcal{F})\big) \vdash \mathsf{u}(\mathcal{F})$$

The proof breaks into two cases. The first reduces to $\langle x = \texttt{null}\rangle \wedge (\triangleright \cdots) \vdash \mathsf{u}(\mathcal{F})$, which follows from $\mathsf{uniform/emp}$ rule. The second case reduces to $\big(x \mapsto (d, l, r) \star \mathsf{tree}(l) \star \mathsf{tree}(r)\big) \wedge \big(\triangleright \circledcirc\forall x.\mathsf{tree}(x) \Rightarrow \mathsf{u}(\mathcal{F})\big) \vdash \mathsf{u}(\mathcal{F})$. Then the $\mathsf{uniform}\star$ rule gives

$$\big(x \mapsto (d, l, r) \star (\mathsf{tree}(l) \star \mathsf{tree}(r))\big) \wedge \big(\triangleright \circledcirc\forall x.\mathsf{tree}(x) \Rightarrow \mathsf{u}(\mathcal{F})\big) \vdash \mathsf{u}(\mathcal{F}) \star \mathsf{u}(\mathcal{F})$$

We now can cut with the $\triangleright_\pi\star$ rule to meet the inductive guard since $x \mapsto (d, l, r) \vdash \mathsf{uniform}(\mathcal{F}) \wedge \neg\mathsf{emp}$ due to the rules $\mapsto\mathsf{uniform}$ and $\mapsto\mathsf{emp}$. Our remaining goal is thus

$$\big(x \mapsto (d, l, r) \wedge \triangleright \cdots\big) \star \big((\mathsf{tree}(l) \star \mathsf{tree}(r)) \wedge \circledcirc\forall x.\mathsf{tree}(x) \Rightarrow \mathsf{u}(\mathcal{F})\big) \vdash \mathsf{u}(\mathcal{F}) \star \mathsf{u}(\mathcal{F})$$

We split over $\star$. The first goal is $x \mapsto (d, l, r) \wedge \triangleright \cdots \vdash \mathsf{u}(\mathcal{F})$, which follows from $\mapsto\mathsf{u}$. The second goal is $(\mathsf{tree}(l) \star \mathsf{tree}(r)) \wedge \circledcirc\forall x.\mathsf{tree}(x) \Rightarrow \mathsf{u}(\mathcal{F})\big) \vdash \mathsf{u}(\mathcal{F})$. We apply $\circledcirc\star$ to distribute the inductive hypothesis into the $\star$, and $\mathsf{uniform}\star$ to split the right hand side, yielding

$$\big(\mathsf{tree}(l) \wedge \circledcirc\forall x.\mathsf{tree}(x) \Rightarrow \mathsf{u}(\mathcal{F})\big) \star \big(\mathsf{tree}(r) \wedge \circledcirc\forall x.\mathsf{tree}(x) \Rightarrow \mathsf{u}(\mathcal{F})\big) \vdash \mathsf{u}(\mathcal{F}) \star \mathsf{u}(\mathcal{F})$$

We again split over $\star$ to reach two essentially identical cases. We apply rule T to remove the $\circledcirc$ and then reach *e.g.* $\forall x.\mathsf{tree}(x) \Rightarrow \mathsf{u}(\mathcal{F}) \vdash \mathsf{tree}(l) \Rightarrow \mathsf{u}(\mathcal{F})$, which is immediate. Further details on this proof can be found in the full paper [2].

Proving that $\mathsf{list}(x)$ *is precise.* Precision is more complex than π-uniformity, so it is harder to prove. We will use the simpler $\mathsf{list}(x)$ as an example; the additional trick we need to prove that $\mathsf{tree}(x)$ is precise are applications of the $\triangleright_\pi\circledcirc$ and $\circledcirc\star$ rules in the same manner as the proof that $\mathsf{tree}(x)$ is $\mathcal{F}$-uniform. We have proved that both $\mathsf{list}(x)$ and $\mathsf{tree}(x)$ are precise using our proof rules in Coq [1].

$$\frac{}{\mathsf{precisely}(P) \dashv\vdash (P \star \top) \Rightarrow \mathsf{precisely}(P)}\ (A) \qquad \frac{\mathrm{precise}(P)}{P \star \mathsf{precisely}(Q) \vdash \mathsf{precisely}(P \star Q)}\ (D)$$

$$\frac{Q \wedge (R \star \top) \vdash \mathsf{precisely}(R) \quad Q \wedge (S \star \top) \vdash \mathsf{precisely}(S) \quad (R \star \top) \wedge (S \star \top) \vdash \bot}{Q \wedge \big((R \vee S) \star \top\big) \vdash \mathsf{precisely}\big(R \vee S\big)}\ (B)$$

$$\frac{\forall x.\Big(Q \wedge \big(P(x) \star \top\big) \vdash \mathsf{precisely}\big(P(x)\big)\Big) \quad \forall x, y.\Big(\big(P(x) \star \top\big) \wedge \big(P(y) \star \top\big) \vdash \lfloor x = y \rfloor\Big)}{Q \wedge \Big(\big(\exists x.P(x)\big) \star \top\Big) \vdash \mathsf{precisely}\Big(\exists x.P(x)\Big)}\ (C)$$

Fig. 10. Key lemmas we use to prove recursive predicates precise

In Fig. 10 we give four key lemmas used in our proof[2]. All four are derived (with a little cleverness) from the proof rules given in Fig. 8. We sketch the proof as follows. To prove precise(list(x)) we first use the preciselyPRECISE rule to transform the goal into $\top \vdash \mathsf{precisely}(\mathsf{list}(x))$. We cannot immediately apply rule W, however, since without a concrete $\star$-separated conjunct **outside** the precisely, we cannot dismiss the inductive guard with the $\rhd_\pi\star$ rule. Accordingly, we next use lemma (A) and standard natural deduction to reach the goal $\top \vdash \forall x.(\mathsf{list}(x) \star \top) \Rightarrow \mathsf{precisely}(\mathsf{list}(x))$, after which we apply rule W with $\pi = \mathcal{F}$.

Afterwards we do some standard natural deduction steps yielding the goal

$$\Big(\rhd \forall x.\big(\mathsf{list}(x) \star \top\big) \Rightarrow \mathsf{precisely}\big(\mathsf{list}(x)\big)\Big) \wedge \Big(\big(\langle x = \mathtt{null}\rangle \vee \exists d, n.x \mapsto (d, n) \star \mathsf{list}(n)\big) \star \top\Big) \vdash \mathsf{precisely}\big(\langle x = \mathtt{null}\rangle \vee \exists d, n.x \mapsto (d, n) \star \mathsf{list}(n)\big)$$

We are now in a position to apply lemma (B) to break up the conjunction. We now have three goals. The first goal is that $\langle x = \mathtt{null}\rangle$ is precise, which follows from the fact that emp is precise, which in turn can be proved using the rule preciselyRIGHT. The third goal is that the two branches of the disjunction are mutually incompatible, which follows from $\langle x = \mathtt{null}\rangle$ being incompatible with maps-to using rule $\mapsto$null. The second (and last remaining) goal needs to use lemma (C) twice to break up the existentials. Two of the three new goals are to show that the two existentials are uniquely determined, which follow from $\mapsto$INVERSION, leaving the goal

$$\Big(\rhd \forall x.\big(\mathsf{list}(x) \star \top\big) \Rightarrow \mathsf{precisely}\big(\mathsf{list}(x)\big)\Big) \wedge \Big(x \mapsto (d, n) \star \big(\mathsf{list}(n) \star \top\big)\Big) \vdash \mathsf{precisely}\Big(x \mapsto (d, n) \star \mathsf{list}(n)\Big)$$

We now cut with lemma (D), using rule $\mapsto$PRECISE to prove its premise, yielding

$$\Big(\rhd \forall x.\big(\mathsf{list}(x) \star \top\big) \Rightarrow \mathsf{precisely}\big(\mathsf{list}(x)\big)\Big) \wedge \Big(x \mapsto (d, n) \star \big(\mathsf{list}(n) \star \top\big)\Big) \vdash x \mapsto (d, n) \star \mathsf{precisely}\Big(\mathsf{list}(n)\Big)$$

We now use $\rhd_\pi\star$ rule to defeat the inductive guard. The rest is straightforward. Further details on this proof can be found in the full paper [2].

6 The ShareInfer fractional biabduction engine

Having described our logical machinery in Sects. 3, 4 and 5, we now demonstrate that our techniques are well-suited to automation by documenting our ShareInfer

[2] We abuse notation by reusing the inference rule format to present derived lemmas.

Precision		Uniformity		Bi-abduction	
File name	Time (ms)	File name	Time (ms)	File name	Time (ms)
precise_map1	0.1	uni_map1	0.2	bi_map1	1.3
precise_map2	0.2	uni_map2	0.8	bi_map2	0.9
precise_map3	1.2	uni_map3	0.3	bi_map3	0.5
precise_list1	2.7	uni_list1	1.2	bi_list1	4.0
precise_list2	1.3	uni_list2	2.1	bi_list2	3.2
precise_list3	3.4	uni_list3	0.7	bi_list3	3.8
precise_tree1	1.4	uni_tree1	1.9	bi_tree1	5.1
precise_tree2	1.7	uni_tree2	1.0	bi_tree2	6.5
precise_tree3	12.2	uni_tree3	10.3	bi_tree3	7.9

Fig. 11. Evaluation of our proof systems using ShareInfer

prototype [1]. Our tool is capable of checking whether a user-defined recursive predicate such as list or tree is uniform and/or precise and then conducting biabductive inference over a separation logic entailment containing said predicates.

To check uniformity, the tool first uses heuristics to guess a potential tree share candidate π and then applies proof rules in Figs. 7 and 6 to derive the goal $\mathsf{uniform}(\pi)$. To support more flexibility, our tool also allows users to specify the candidate share π manually. To check precision, the tool maneuvers over the proof rules in Figs. 6 and 8 to achieve the desired goal. In both cases, recursive predicates are handled with the rules in Fig. 9. ShareInfer returns either Yes, No or Unknown together with a human-readable proof of its claim.

For bi-abduction, ShareInfer automatically checks precision and uniformity whenever it encounters a new recursive predicate. If the check returns Yes, the tool will unlock the corresponding rule, *i.e.*, DotPlus for precision and DotStar for uniformity. ShareInfer then matches fragments between the consequent and antecedent while applying folding and unfolding rules for recursive predicates to construct the antiframe and inference frame respectively. For instance, here is the biabduction problem contained in file bi_tree2 (see Fig. 11):

$$a \overset{\mathcal{F}}{\longmapsto} (b, c, d) \ \star \ \mathcal{L} \cdot \mathsf{tree}(c) \ \star \ \mathcal{R} \cdot \mathsf{tree}(d) \ \star \ [??] \ \vdash \ \mathcal{L} \cdot \mathsf{tree}(a) \ \star \ [??]$$

ShareInfer returns antiframe $\mathcal{L}\cdot\mathsf{tree}(d)$ and inference frame $a \overset{\mathcal{R}}{\longmapsto} (b, c, d) \star \mathcal{R}\cdot\mathsf{tree}(d)$.

ShareInfer is around 2.5k LOC of Java. We benchmarked it with 27 selective examples from three categories: precision, uniformity and bi-abduction. The benchmark was conducted with a 3.4 GHz processor and 16 GB of memory. Our results are given in Fig. 11. Despite the complexity of our proof rules our performance is reasonable: ShareInfer only took 75.9 ms to run the entire example set, or around 2.8 ms per example. Our benchmark is small, but this performance indicates that more sophisticated separation logic verifiers such as HIP/SLEEK [14] or Infer [9] may be able to use our techniques at scale.

7 Building a Model for Our Logic

Our task now is to provide a model for our proof theories. We present our models in several parts. In Sect. 7.1 we begin with a brief review of Cancellative Separation Algebras (CSA). In Sect. 7.2 we explain what we need from our fractional share models. In Sect. 7.3 we develop an extension to CSAs called "Scaling Separation Algebras" (SSA). In Sect. 7.5 we develop the machinery necessary to support our rules for object-level induction over the heap. We have verified in Coq [1] that the models in Sect. 7.1 support the rules in Fig. 8, the models in Sect. 7.3 support the rules Figs. 3 and 7, and the models in Sect. 7.5 support the rules in Fig. 9.

7.1 Cancellative Separation Algebras

A Separation Algebra (SA) is a set H with an associative, commutative partial operation $\oplus$. Separation algebras can have a single unit or multiple units; we use $\mathit{identity}(x)$ to indicate that x is a unit. A Cancellative SA $\langle H, \oplus \rangle$ further requires that $a \oplus b_1 = c \Rightarrow a \oplus b_2 = c \Rightarrow b_1 = b_2$. We can define a partial order on H using $\oplus$ by $h_1 \subseteq h_2 \overset{\text{def}}{=} \exists h'. h_1 \oplus h' = h_2$. Calcagno *et al.* [12] showed that CSAs can model separation logic with the definitions

$$h \models P \star Q \overset{\text{def}}{=} \exists h_1, h_2.\ h_1 \oplus h_2 = h \wedge (h_1 \models P) \wedge (h_2 \models Q) \quad \text{and} \quad h \models \mathsf{emp} \overset{\text{def}}{=} \mathit{identity}(h).$$

The standard definition of precise(P) was given as Eq. (3) in Sect. 5.2, together with the definition for our new $\mathsf{precisely}(P)$ operator in Eq. (4). What is difficult here is finding a set of axioms (Fig. 8) and derivable lemmas (*e.g.* Fig. 10) that are strong enough to be useful in the object-level inductive proofs. Once the axioms are found, proving them from the model given is straightforward. Cancellation is not necessary to model basic separation logic [18], but we need it to prove the introduction preciselyRIGHT and elimination rules preciselyLEFT for our new operator.

7.2 Fractional Share Algebras

A fractional share algebra $\langle S, \oplus, \otimes, \mathcal{E}, \mathcal{F} \rangle$ (FSA) is a set S with two operations: partial addition $\oplus$ and total multiplication $\otimes$. The substructure $\langle S, \oplus \rangle$ is a CSA with the single unit $\mathcal{E}$. For the reasons discussed in Sect. 2 we require that $\oplus$ satisfies the disjointness axiom $a \oplus a = b \Rightarrow a = \mathcal{E}$. Furthermore, we require that the existence of a top element $\mathcal{F}$, representing complete ownership, and assume that each element $s \in S$ has a complement $\overline{s}$ such that $s \oplus \overline{s} = \mathcal{F}$.

Often (*e.g.* in the fractional $\mapsto$ operator) we wish to restrict ourselves to the "positive shares" $S^+ \overset{\text{def}}{=} S \setminus \{\mathcal{E}\}$. To emphasize that a share is positive we often use the metavariable π rather than s. $\oplus$ is still associative, commutative, and cancellative; every element other than $\mathcal{F}$ still has a complement. To enjoy a partial order on S^+ and other SA- or CSA-like structures that lack identities (sometimes called “permission algebras”) we define $\pi_1 \subseteq \pi_2 \overset{\text{def}}{=} (\exists \pi'. \pi_1 \oplus \pi' = \pi_2) \vee (\pi_1 = \pi_2)$.

For the multiplicative structure we require that $\langle S, \otimes, \mathcal{F} \rangle$ be a monoid, *i.e.* that $\otimes$ is associative and has identity $\mathcal{F}$. Since we restrict maps-tos and the permission scaling operator to be positive, we want $\langle S^+, \otimes, \mathcal{F} \rangle$ to be a submonoid. Accordingly, when $\{\pi_1, \pi_2\} \subset S^+$, we require that $\pi_1 \otimes \pi_2 \neq \mathcal{E}$. Finally, we require that $\otimes$ distributes over $\oplus$ on the right, that is $(s_1 \oplus s_2) \otimes s_3 = (s_1 \otimes s_3) \oplus (s_2 \otimes s_3)$; and that $\otimes$ is cancellative on the right given a positive left multiplicand, *i.e.* $\pi \otimes s_1 = \pi \otimes s_2 \Rightarrow s_1 = s_2$.

The tree share model we present in Sect. 2 satisfies all of the above axioms, so we have a nontrivial model. As we will see shortly, it would be very convenient if we could assume that $\otimes$ also distributed on the left, or if we had multiplicative inverses on the left rather than merely cancellation on the right. However, we will see in Sect. 8.2 that both assumptions are untenable.

7.3 Scaling Separation Algebra

A scaling separation algebra (SSA) is $\langle H, S, \oplus_H, \oplus_S, \otimes_S, \mathcal{E}, \mathcal{F}, \mathit{mul}, \mathit{force} \rangle$, where $\langle H, \oplus_H \rangle$ is a CSA for heaps and $\langle S, \oplus_S, \otimes_S, \mathcal{E}, \mathcal{F} \rangle$ is a FSA for shares. Intuitively, $\mathit{mul}(\pi, h_1)$ multiplies every share inside h_1 by π and returns the result h_2. The multiplication is on the left, so for each original share π' in h_1, the resulting share in h_2 is $\pi \otimes_S \pi'$. Recall that the informal meaning of $\pi \cdot P$ is that we have a π-fraction of predicate P. Formally this notion relies on a little trick:

$$h \models \pi \cdot P \overset{\text{def}}{=} \exists h'.\ \mathit{mul}(\pi, h') = \pi \wedge h' \models P \tag{5}$$

A heap h contains a π-fraction of P if there is a **bigger** heap h' satisfying P, and multiplying that bigger heap h' by the scalar π gets back to the smaller heap h.

The simpler $\mathit{force}(\pi, h_1)$ overwrites all shares in h_1 with the constant share π to reach the resulting heap h_2. We use *force* to define the $\mathsf{uniform}$ predicate as $h \models \mathsf{uniform}(\pi) \overset{\text{def}}{=} \mathit{force}(\pi, h) = h$. A heap h is π-uniform when setting all the shares in h to π gets you back to h—*i.e.*, they must have been π to begin with.

$S_1.\ \mathit{force}(\pi, \mathit{force}(\pi', a)) = \mathit{force}(\pi, a)$	$S_2.\ \mathit{force}(\pi, \mathit{mul}(\pi', a)) = \mathit{force}(\pi, a)$
$S_3.\ \mathit{mul}(\pi, \mathit{force}(\pi', a)) = \mathit{force}(\pi \otimes_S \pi', a)$	$S_4.\ \mathit{mul}(\pi, \mathit{mul}(\pi', a)) = \mathit{mul}(\pi \otimes_S \pi', a)$
$S_5.\ \mathit{identity}(a) \Rightarrow \mathit{force}(\pi, a) = a$	$S_6.\ a \subseteq_H \mathit{force}(\mathcal{F}, a)$
$S_7.\ \pi_1 \subseteq_S \pi_2 \Rightarrow \mathit{force}(\pi_1, a) \subseteq_H \mathit{force}(\pi_2, a)$	$S_8.\ \mathit{force}(\pi, a) \oplus_H \mathit{force}(\pi, b) = c \Rightarrow \mathit{force}(\pi, c) = c$
$S_9.\ \mathit{identity}(a) \Rightarrow \mathit{mul}(\pi, a) = a$	$S_{10}.\ \mathit{mul}(\mathcal{F}, a) = a$
$S_{11}.\ \mathit{mul}(\pi, a_1) = \mathit{mul}(\pi, a_2) \Rightarrow a_1 = a_2$	$S_{12}.\ \mathit{mul}(\pi, a) \subseteq_H a$

$S_{13}.\ \pi_1 \oplus_S \pi_2 = \pi_3 \Rightarrow \forall b, c. \big((\mathit{mul}(\pi_1, b) \oplus_H \mathit{mul}(\pi_2, b) = c) \Leftrightarrow (c = \mathit{mul}(\pi_3, b)) \big)$

$S_{14}.\ \mathit{force}(\pi', a) \oplus_H \mathit{force}(\pi', b) = \mathit{force}(\pi', c) \quad \Leftrightarrow$
$\quad \mathit{mul}(\pi, \mathit{force}(\pi', a)) \oplus_H \mathit{mul}(\pi, \mathit{force}(\pi', b)) = \mathit{mul}(\pi, \mathit{force}(\pi', c))$

Fig. 12. The 14 additional axioms for scaling separation algebras beyond those inherited from cancellative separation algebras

We need to understand how all of the ingredients in an SSA relate to each other to prove the core logical rules on page 13. We distill the various relationships we need to model our logic in Fig. 12. Although there are a goodly number of them, most are reasonably intuitive.

Axioms S_1 through S_4 describe how *force* and *mul* compose with each other. Axioms S_5, S_9, and S_{10} give conditions when *force* and *mul* are identity functions: when either is applied to empty heaps, and when *mul* is applied to the multiplicative identity on shares $\mathcal{F}$. Axioms S_6 and S_{12} relate heap order with forcing the full share $\mathcal{F}$ and multiplication by an arbitrary share π. Axiom S_7 says that *force* is order-preserving. Axiom S_8 is how the disjointness axiom on shares is expressed on heaps: when two π-uniform heaps are joined, the result is π-uniform. Axiom S_{11} says that *mul* is injective on heaps. Axiom S_{13} is delicate. In the $\Rightarrow$ direction, it states that *mul* preserves the share model's join structure on heaps. In the $\Leftarrow$ direction, S_{13} is similar to axiom S_8, saying that the share model's join structure **must** be preserved. Taking both directions together, S_{13} translates the **right** distribution property of $\oplus_S$ over $\otimes_S$ into heaps. The final axiom S_{14} is a bit of a compromise. We wish we could satisfy

$$S'_{14}. \qquad a \oplus_H b = c \quad \Leftrightarrow \quad mul(\pi, a) \oplus_H mul(\pi, b) = mul(\pi, c)$$

S'_{14} is a kind of dual for S_{13}, *i.e.* it would correspond to a **left** distributivity property of $\oplus_S$ over $\otimes_S$ in the share model into heaps. Unfortunately, as we will see in Sect. 8.2, the disjointness of $\oplus_S$ is incompatible with simultaneously supporting both left and right distributivity. Accordingly, S_{14} weakens S'_{14} so that it only holds when a and b are π'-uniform (which by S_8 forces c to be π'-uniform). We also wish we could satisfy S'_{15}: $\forall \pi, a. \exists b. mul(\pi, b) = a$, which corresponds to left multiplicative inverses, but again (Sect. 8.2) disjointness is incompatible.

7.4 Compositionality of Scaling Separation Algebras

Despite their complex axiomatization, we gain two advantages from developing SSAs rather than directly proving our logical axioms on a concrete model. First, they give us a precise understanding of exactly which operations and properties (S_1–S_{14}) are used to prove the logical axioms. Second, following Dockins *et al.* [21] we can build up large SSAs compositionally from smaller SSAs.

To do so cleanly it will be convenient to consider a slight variant of SSAs, "Weak SSAs" that allow, but do not require, the existence of identity elements in the underlying CSA model. A WSSA satisfies exactly the same axioms as an SSA, except that we use the weaker $\subseteq_H$ definition we defined for permission algebras, *i.e.* $a_1 \subseteq_H a_2 \overset{\text{def}}{=} (\exists a'. a_1 \oplus_H a' = a_2) \vee (a_1 = a_2)$. Note that S_5 and S_9 are vacuously true when the CSA does not have identity elements. We need identity elements to prove the logical axioms from the model; we only use WSSAs to gain compositionality as we construct a suitable final SSA. Keeping the share components $\langle S, \oplus_S, \otimes_S, \mathcal{E}, \mathcal{F} \rangle$ constant, we give three SSA constructors to get a flavor for what we can do with the remaining components $\langle H, \oplus_H, force, mul \rangle$.

Example 1 (Shares). The share model $\langle S, \oplus_S \rangle$ is an SSA, and the positive (non-$\mathcal{E}$) shares $\langle S^+, \oplus \rangle$ are a WSSA, with $\mathit{force}_S(\pi, \pi') \stackrel{\text{def}}{=} \pi$ and $\mathit{mul}_S(\pi, \pi') \stackrel{\text{def}}{=} \pi \otimes \pi'$.

Example 2 (Semiproduct). Let $\langle A, \oplus_A, \mathit{force}_A, \mathit{mul}_A \rangle$ be an SSA/WSSA, and B be a set. Define $(a_1, b_1) \oplus_{A \times B} (a_2, b_2) = (a_3, b_3) \stackrel{\text{def}}{=} a_1 \oplus_A a_2 = a_3 \wedge b_1 = b_2 = b_3$, $\mathit{force}_{A \times B}(\pi, (a, b)) \stackrel{\text{def}}{=} (\mathit{force}_A(\pi, a), b)$, and $\mathit{mul}_{A \times B}(\pi, (a, b)) \stackrel{\text{def}}{=} (\mathit{mul}_A(\pi, a), b)$. Then $\langle A \times B, \oplus_{A \times B}, \mathit{force}_{A \times B}, \mathit{mul}_{A \times B} \rangle$ is an SSA/WSSA.

Example 3 (Finite partial map). Let A be a set and $\langle B, \oplus_B, \mathit{force}_B, \mathit{mul}_B \rangle$ be an SSA/WSSA. Define $f \oplus_{A \stackrel{\mathsf{fin}}{\rightharpoonup} B} g = h$ pointwise [21]. Define $\mathit{force}_{A \stackrel{\mathsf{fin}}{\rightharpoonup} B}(\pi, f) \stackrel{\text{def}}{=} \lambda x. \mathit{force}_B(\pi, f(x))$ and likewise define $\mathit{mul}_{A \stackrel{\mathsf{fin}}{\rightharpoonup} B}(\pi, f) \stackrel{\text{def}}{=} \lambda x. \mathit{mul}_B(\pi, f(x))$. The structure $\langle A \stackrel{\mathsf{fin}}{\rightharpoonup} B, \oplus_{A \stackrel{\mathsf{fin}}{\rightharpoonup} B}, \mathit{force}_{A \stackrel{\mathsf{fin}}{\rightharpoonup} B}, \mathit{mul}_{A \stackrel{\mathsf{fin}}{\rightharpoonup} B} \rangle$ is an SSA.

Using these constructors, $A \stackrel{\mathsf{fin}}{\rightharpoonup} (S^+, V)$, *i.e.* finite partial maps from addresses to pairs of positive shares and values, is an SSA and thus can support a model for our logic. We also support other standard constructions *e.g.* sum types $+$.

7.5 Model for Inductive Logic

What remains is to give the model that yields the inductive logic in Fig. 9. The key induction guard modal $\triangleright_\pi$ operator is defined as follows:

$$\begin{array}{lcl} h_1 \ S_\pi \ h_4 & \stackrel{\text{def}}{=} & \exists h_2, h_3.\ h_1 \supseteq_H h_2 \wedge h_3 \oplus_H h_4 = h_2 \wedge (h_3 \models \mathsf{uniform}(\pi) \wedge \neg \mathsf{emp}) \\ h \models \triangleright_\pi P & \stackrel{\text{def}}{=} & \forall h'.\ (h \ S_\pi \ h') \Rightarrow (h' \models P) \end{array}$$

In other words, $\triangleright_\pi$ is a (boxy) modal operator over the relation S_π, which relates a heap h_1 with all heaps that are strict subheaps that are smaller by at least a π-piece. The model is a little subtle to enable the rules $\triangleright_\pi \circledcirc$ and $\circledcirc \triangleright_\pi$ that let us handle multiple recursive calls and simplify the engineering. The within operator $\circledcirc$ is much simpler to model:

$$h_1 \ W \ h_2 \ \stackrel{\text{def}}{=} \ h_1 \supseteq_H h_2 \qquad\qquad h \models \circledcirc P \ \stackrel{\text{def}}{=} \ \forall h'.\ (h \ W \ h') \Rightarrow (h' \models P)$$

All of the rules in Fig. 9 follow from these definitions except for rule W. To prove this rule, we require that the heap model have an additional operator. The "π-quantum", written $|h|_\pi$, gives the number of times a non-empty π-sized piece can be taken out of h. For disjoint shares, the number of times is no more than the number of defined memory locations in h. We require two facts for $|h|_\pi$. First, that $h_1 \subseteq_H h_2 \Rightarrow |h_1|_\pi \leq |h_2|_\pi$, *i.e.* that subheaps do not have larger π-quanta than their parent. Second, that $h_1 \oplus_H h_2 = h_3 \Rightarrow (h_2 \models \mathsf{uniform}(\pi) \wedge \neg \mathsf{emp}) \Rightarrow |h_3|_\pi > |h_1|_\pi$, *i.e.* that taking out a π-piece strictly decreases the number of π-quanta. Given this setup, rule W follows immediately by induction on $|h|_\pi$. The rules that require the longest proofs in the model are $\triangleright_\pi \circledcirc$ and $\circledcirc \triangleright_\pi$.

8 Lower Bounds on Predicate Multiplication

In Sect. 7 we gave a model for the logical axioms we presented in Fig. 3 and on page 13. Our goal here is to show that it is difficult to do better, *e.g.* by having a premise-free DotStar rules or a bidirectional DotImpl rule. In Sect. 8.1 we show that these logical rules force properties on the share model. In Sect. 8.2 we show that disjointness puts restrictions on the class of share models. There are no non-trivial models that have left inverses or satisfy both left and right distributivity.

8.1 Predicate Multiplication's Axioms Force Share Model Properties

The SSA structures we gave in Sect. 7.3 are good for building models that enable the rules for predicate multiplication from Fig. 3. However, since they impose intermediate algebraic and logical signatures between the concrete model and rules for predicate multiplication, they are not good for showing that we cannot do better. Accordingly here we disintermediate and focus on the concrete model $A \overset{\mathsf{fin}}{\rightharpoonup} (S^+, V)$, that is finite partial maps from addresses to pairs of positive shares and values. The join operations on heaps operates pointwise [21], with $(\pi_1, v_1) \oplus (\pi_2, v_2) = (\pi_3, v_3) \overset{\text{def}}{=} \pi_1 \oplus_S \pi_2 = \pi_3 \wedge v_1 = v_2 = v_3$, from which we derive the usual SA model for $\star$ and emp (Sect. 7.1). We define $h \models x \overset{\pi}{\mapsto} y \overset{\text{def}}{=} dom(h) = \{x\} \wedge h(x) = (\pi, y)$. We define scalar multiplication over heaps $\otimes_H$ pointwise as well, with $\pi_1 \otimes (\pi_2, v) \overset{\text{def}}{=} (\pi_1 \otimes_S \pi_2, v)$, and then define predicate multiplication by $h \models \pi \cdot P \overset{\text{def}}{=} \exists h'.\ h' = \pi \otimes_H h' = h \wedge h' \models P$. All of the above definitions are standard except for $\otimes_H$, which strikes us as the only choice (up to commutativity), and predicate multiplication itself.

By Sect. 7 we already know that this model satisfies the rules for predicate multiplication, given the assumptions on the share model from Sect. 7.2. What is interesting is that we can prove the other direction: if we assume that the key logical rules from Fig. 3 hold, they force axioms on the share model. The key correspondences are: DotFull forces that $\mathcal{F}$ is the left identity of $\otimes_S$; DotMapsTo forces that $\mathcal{F}$ is the right identity of $\otimes_S$; DotMapsTo forces the associativity of $\otimes_S$; the $\dashv$ direction of DotConj forces the right cancellativity of $\otimes_S$ (as does DotImpl and the $\dashv$ direction of DotUniv); and DotPlus, which forces right distributivity of $\otimes_S$ over $\oplus_S$.

The following rules force left distributivity of $\otimes_S$ over $\oplus_S$ and left $\otimes_S$ inverses:

$$\frac{}{\pi \cdot (P \star Q) \dashv\vdash (\pi \cdot P) \star (\pi \cdot Q)}\ \text{DotStar}' \qquad \frac{}{\pi \cdot (P \Rightarrow Q) \dashv (\pi \cdot P) \Rightarrow (\pi \cdot Q)}\ \text{DotImpl}'$$

The $\dashv$ direction of DotStar′ also forces that $\oplus_S$ satisfies disjointness; this is the key reason that we cannot use rationals $\langle (0, 1], +, \times \rangle$. Clearly the side-condition-free DotStar′ rule is preferable to the DotStar in Fig. 3, and it would also be preferable to have bidirectionality for predicate multiplication over implication and negation. Unfortunately, as we will see shortly, the disjointness of $\oplus_S$ places strong multiplicative algebraic constraints on the share model. These constraints are the reason we cannot support the DotImpl$'$ rule and why we require the π'-uniformity side condition in our DotStar rule.

8.2 Disjointness in a Multiplicative Setting

Our goal now is to explore the algebraic consequences of the disjointness property in a multiplicative setting. Suppose $\langle S, \oplus \rangle$ is a CSA with a single unit $\mathcal{E}$, top element $\mathcal{F}$, and $\oplus$ complements $\overline{s}$. Suppose further that shares satisfy the disjointness property $a \oplus a = b \Rightarrow a = \mathcal{E}$. For the multiplicative structure, assume $\langle S, \otimes, \mathcal{F} \rangle$ is a monoid (*i.e.* the axioms forced by the DotDot, DotMapsTo, and DotFull rules). It is undesirable for a share model if multiplying two positive shares (*e.g.* the ability to read a memory cell) results in the empty permission, so we assume that when π_1 and π_2 are non-$\mathcal{E}$ then their product $\pi_1 \otimes \pi_2 \neq \mathcal{E}$.

Now add left or right distributivity. We choose right distributivity $(s_1 \oplus s_2) \otimes s_3 = (s_1 \otimes s_3) \oplus (s_2 \otimes s_3)$; the situation is mirrored with left. Let us show that we cannot have left inverses for $\pi \neq \mathcal{F}$. We prove by contradiction: suppose $\pi \neq \mathcal{F}$ and there exists π^{-1} such that $\pi^{-1} \otimes \pi = \mathcal{F}$. Then

$$\pi = \mathcal{F} \otimes \pi = (\pi^{-1} \oplus \overline{\pi^{-1}}) \otimes \pi = (\pi^{-1} \otimes \pi) \oplus (\overline{\pi^{-1}} \otimes \pi) = \mathcal{F} \oplus (\overline{\pi^{-1}} \otimes \pi)$$

Let $e = \overline{\pi^{-1}} \otimes \pi$. Now $\pi = \mathcal{F} \oplus e = (\overline{e} \oplus e) \oplus e$, which by associativity and disjointness forces $e = \mathcal{E}$, which in turn forces $\pi = \mathcal{F}$, a contradiction.

Now suppose that instead of adding multiplicative inverses we have both left and right distributivity. First we prove (Lemma 1) that for arbitrary $s \in S$, $s \otimes \overline{s} = \overline{s} \otimes s$. We calculate:

$$(s \otimes s) \oplus (s \otimes \overline{s}) = s \otimes (s \oplus \overline{s}) = s \otimes \mathcal{F} = s = \mathcal{F} \otimes s = (s \oplus \overline{s}) \otimes s = (s \otimes s) \oplus (\overline{s} \otimes s)$$

Lemma 1 follows by the cancellativity of $\oplus$ between the far left and the far right.

Now we show (Lemma 2) that $s \otimes \overline{s} = \mathcal{E}$. We calculate:

$$\begin{aligned}\mathcal{F} = \mathcal{F} \otimes \mathcal{F} = (s \oplus \overline{s}) \otimes (s \oplus \overline{s}) &= (s \otimes s) \oplus (s \otimes \overline{s}) \oplus (\overline{s} \otimes s) \oplus (\overline{s} \otimes \overline{s}) \\ &= (s \otimes s) \oplus \underline{(s \otimes \overline{s}) \oplus (s \otimes \overline{s})} \oplus (\overline{s} \otimes \overline{s})\end{aligned}$$

The final equality is by Lemma 1. The underlined portion implies $s \otimes \overline{s} = \mathcal{E}$ by disjointness. The upshot of Lemma 2, together with our requirement that the product of two positive shares be positive, is that we can have no more than the two elements $\mathcal{E}$ and $\mathcal{F}$ in our share model. Since the entire motivation for fractional share models is to allow ownership between $\mathcal{E}$ and $\mathcal{F}$, we must choose either left or right distributivity; we choose right since we are able to prove that the π'-uniformity side condition enables the bidirectional DotStar.

9 Related Work

Fractional permissions are essentially used to reason about resource ownership in concurrent programming. The well-known rational model $\langle [0, 1], + \rangle$ by Boyland *et al.* [5] is used to reason about join-fork programs. This structure has the disjointness problem mentioned in Sect. 2, first noticed by Bornat *et al.* [4], as well as other problems discussed in Sects. 3, 4, and [2]. Boyland [6] extended the framework to scale permissions uniformly over arbitrary predicates with multiplication, *e.g.*, he defined $\pi \cdot P$ as "multiply each permission π' in P with π". However, his framework cannot fit into SL and his scaling rules are not bi-directional. Jacobs and Piessens [28] also used rationals for scaling permissions $\pi \cdot P$ in SL but only obtained one direction for DotStar and DotPlus. A different kind of scaling permission was used by Dinsdale-Young *et al.* [20] in which they used rationals to define permission assertions $[A]^r_\pi$ to indicate a thread with permission π can execute the action A over the shared region r.

There are other flavors of permission besides rationals. Bornat *et al.* [4] introduced integer counting permissions $\langle \mathbb{Z}, +, 0 \rangle$ to reason about semaphores and combined rationals and integers into a hybrid permission model. Heule *et al.* [23] flexibly allowed permissions to be either concretely rational or abstractly read-only to lower the nuisance of detailed accounting. A more general read-only permissions was proposed by Charguéraud and Pottier [13] that transforms a predicate P into read-only mode $\mathsf{RO}(P)$ which can duplicated/merged with the bi-entailment $\mathsf{RO}(P) \dashv\vdash \mathsf{RO}(P) \star \mathsf{RO}(P)$. Their permissions distribute pleasantly over disjunction and existential quantifier but only work one way for $\star$, *i.e.*, $\mathsf{RO}(H_1 \star H_2) \vdash \mathsf{RO}(H_1) \star \mathsf{RO}(H_2)$. Parkinson [41] proposed subsets of the natural numbers for shares $\langle \mathcal{P}(\mathbb{N}), \uplus \rangle$ to fix the disjointness problem. Compared to tree shares, Parkinson's model is less practical computationally and does not have an obvious multiplicative structure.

Protocol-based logics like FCSL [38] and Iris [30] have been very successful in reasoning about fine-grained concurrent programs, but their high expressivity results in a heavyweight logic. Automation (*e.g.* inference such as we do in Sect. 4) has been hard to come by. We believe that fractional permissions and protocol-based logics are in a meaningful sense complementary rather than competitors.

Verification tools often implement rational permissions because of its simplicity. For example, VeriFast [29] uses rationals to verify programs with locks and semaphores. It also allows simple and restrictive forms of scaling permissions which can be applied uniformly over standard predicates. On the other hand, HIP/SLEEK [31] uses rationals to model "thread as resource" so that the ownership of a thread and its resources can be transferred. Chalice [36] has rational permissions to verify properties of multi-threaded, objected-based programs such as data races and dead-locks. Viper [37] has an expressive intermediate language that supports both rational and abstract permissions. However, a number of verification tools have chosen tree shares due to their better metatheoretical properties. VST [3] is equipped with tree share permissions and an extensive tree share library. HIP/SLEEK uses tree shares to verify the barrier structure [26] and has its own complete share solver [33,35] that reduces tree formulae to Boolean formulae handled by Z3 [17]. Lastly, tree share permissions are featured in Heap-Hop [47] to reason over asynchronous communications.

10 Conclusion

We presented a separation logic proof framework to reason about resource sharing using fractional permissions in concurrent verification. We support sophisticated verification tasks such as inductive predicates, proving predicates precise, and biabduction. We wrote ShareInfer to gauge how our theories could be automated. We developed scaling separation algebras as compositional models for our logic. We investigated why our logic cannot support certain desirable properties.

References

1. http://www.comp.nus.edu.sg/~lxbach/tools/share_infer/
2. http://www.comp.nus.edu.sg/~lxbach/publication/permission_full.pdf
3. Appel, A.W.: Verified software toolchain. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 1–17. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19718-5_1
4. Bornat, R., Cristiano, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL (2005)
5. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_4
6. Boyland, J.T.: Semantics of fractional permissions with nesting. ACM Trans. Program. Lang. Syst. **32**(6), 22 (2010)
7. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: POPL (2008)
8. Brotherston, J., Gorogiannis, N., Kanovich, M.: Biabduction (and related problems) in array separation logic. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 472–490. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_29
9. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 3–11. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_1
10. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL (2009)
11. Calcagno, C., Distefano, D., Vafeiadis, V.: Bi-abductive resource invariant synthesis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 259–274. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10672-9_19
12. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS, pp. 366–378 (2007)
13. Charguéraud, A., Pottier, F.: Temporary read-only permissions for separation logic. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 260–286. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_10
14. Chin, W.-N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. **77**(9), 1006–1036 (2012)
15. Chin, W.N., Le, T.C., Qin, S.: Automated verification of countdownlatch (2017)

16. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: a logic for time and data abstraction. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 207–231. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44202-9_9
17. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
18. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., Yang, H.: Views: compositional reasoning for concurrent programs. In: POPL (2013)
19. Dinsdale-Young, T., Pinto, P.D.R., Andersen, K.J., Birkedal, L.: Caper: automatic verification for fine-grained concurrency. In: ESOP (2017)
20. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_24
21. Dockins, R., Hobor, A., Appel, A.W.: A fresh look at separation algebras and share accounting. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 161–177. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10672-9_13
22. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: PLDI (2007)
23. Heule, S., Leino, K.R.M., Müller, P., Summers, A.J.: Fractional permissions without the fractions. In: FTfJP, pp. 1:1–1:6 (2011)
24. Hobor, A.: Oracle Semantics. Ph.D. thesis, Department of Computer Science, Princeton University, Princeton, October 2008
25. Hobor, A., Gherghina, C.: Barriers in concurrent separation logic. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 276–296. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19718-5_15
26. Hobor, A., Gherghina, C.: Barriers in concurrent separation logic: now with tool support! Log. Methods Comput. Sci. 8(2) (2012)
27. Hoenicke, J., Majumdar, R., Podelski, A.: Thread modularity at many levels: a pearl in compositional verification. In: POPL (2017)
28. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. In: POPL (2011)
29. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the verifast program verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17164-2_21
30. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In: POPL (2015)
31. Le, D.-K., Chin, W.-N., Teo, Y.M.: Threads as resource for concurrency verification. In: PEPM (2015)
32. Le, Q.L., Gherghina, C., Qin, S., Chin, W.-N.: Shape analysis via second-order bi-abduction. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 52–68. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_4
33. Le, X.B., Gherghina, C., Hobor, A.: Decision procedures over sophisticated fractional permissions. In: Jhala, R., Igarashi, A. (eds.) APLAS 2012. LNCS, vol. 7705, pp. 368–385. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35182-2_26
34. Le, X.-B., Hobor, A., Lin, A.W.: Decidability and complexity of tree shares formulas. In: FSTTCS (2016)

35. Le, X.-B., Nguyen, T.-T., Chin, W.-N., Hobor, A.: A certified decision procedure for tree shares. In: Duan, Z., Ong, L. (eds.) ICFEM 2017. LNCS, vol. 10610, pp. 226–242. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68690-5_14
36. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 378–393. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00590-9_27
37. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
38. Nanevski, A., Ley-Wild, R., Sergey, I., Delbianco, G.A.: Communicating state transition systems for fine-grained concurrent resources. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 290–310. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_16
39. Nguyen, H.H., David, C., Qin, S., Chin, W.-N.: Automated verification of shape and size properties via separation logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69738-1_18
40. OHearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. **375**(1–3), 271–307 (2007)
41. Parkinson, M.: Local Reasoning for Java. Ph.D. thesis, University of Cambridge (2005)
42. Parkinson, M.J., Bornat, R., O'Hearn, P.W.: Modular verification of a non-blocking stack. In: POPL (2007)
43. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: LICS (2002)
44. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: PLDI (2015)
45. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis (2008)
46. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74407-8_18
47. Villard, J.: Heaps and Hops. Ph.D. thesis, Laboratoire Spécification et Vérification, École Normale Supérieure de Cachan, France, February 2011
48. Windsor, M., Dodds, M., Simner, B., Parkinson, M.J.: Starling: lightweight concurrency verification with views. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 544–569. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_27
49. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_36

Deadlock-Free Monitors

Jafar Hamin(✉) and Bart Jacobs

imec-DistriNet, Department of Computer Science, KU Leuven,
Celestijnenlaan 200A, 3001 Heverlee, Belgium
{jafar.hamin,bart.jacobs}@cs.kuleuven.be

Abstract. Monitors constitute one of the common techniques to synchronize threads in multithreaded programs, where calling a wait command on a condition variable suspends the caller thread and notifying a condition variable causes the threads waiting for that condition variable to resume their execution. One potential problem with these programs is that a waiting thread might be suspended forever leading to deadlock, a state where each thread of the program is waiting for a condition variable or a lock. In this paper, a modular verification approach for deadlock-freedom of such programs is presented, ensuring that in any state of the execution of the program if there are some threads suspended then there exists at least one thread running. The main idea behind this approach is to make sure that for any condition variable v for which a thread is waiting there exists a thread obliged to fulfil an obligation for v that only waits for a waitable object whose wait level, an arbitrary number associated with each waitable object, is less than the wait level of v. The relaxed precedence relation introduced in this paper, aiming to avoid cycles, can also benefit some other verification approaches, verifying deadlock-freedom of other synchronization constructs such as channels and semaphores, enabling them to accept a wider range of deadlock-free programs. We encoded the proposed proof rules in the VeriFast program verifier and by defining some appropriate invariants for the locks associated with some condition variables succeeded in verifying some popular use cases of monitors including unbounded/bounded buffer, sleeping barber, barrier, and readers-writers locks. A soundness proof for the presented approach is provided; some of the trickiest lemmas in this proof have been machine-checked with Coq.

1 Introduction

One of the popular mechanisms for synchronizing threads in multithreaded programs is using monitors, a synchronization construct allowing threads to have mutual exclusion and also the ability to wait for a certain condition to become true. These constructs, consisting of a mutex/lock and some condition variables, provide some basic functions for their clients, namely $\mathsf{wait}(v, l)$, causing the calling thread to wait for the condition variable v and release lock l while doing so, and $\mathsf{notify}(v)$/$\mathsf{notifyAll}(v)$, causing one/all thread(s) waiting for v to resume their execution. Each condition variable is associated with a lock; a thread must

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 415–441, 2018.
https://doi.org/10.1007/978-3-319-89884-1_15

acquire the associated lock for waiting or notifying on a condition variable, and when a thread is notified it must reacquire the associated lock.

However, one potential problem with these synchronizers is deadlock, where all threads of the program are waiting for a condition variable or a lock. To clarify the problem consider the program in Fig. 1, where a channel consists of a queue q, a lock l and a condition variable v, protecting a thread from dequeuing q when it is empty. In this program the receiver thread first acquires lock l and while there is no item in q it releases l, suspends itself and waits for a notification on v. If this thread is notified while q is not empty it dequeues an item and finally releases l. The sender thread also acquires the same lock, enqueues an item into q, notifies one of the threads waiting for v, if any, and lastly releases l. After creating a channel ch, the main thread of the program first forks a thread to receive a message from ch and then sends a message on ch. Although this program is deadlock-free, it is easy to construct some variations of it that lead to deadlock: if the main thread itself, before sending any messages, tries to receive a message from ch, or if the number of receives is greater than the number of sends, or if the receiver thread waits for v even if q is not empty.

```
routine main()
{q := newqueue;
 l := newlock;
 v := newcond;
 ch := channel(q, l, v);
 fork (receive(ch));
 send(ch, 12)}

routine send(channel ch, int d)
{acquire(ch.l);
 enqueue(ch.q, d);
 notify(ch.v);
 release(ch.l)}

routine receive(channel ch)
{acquire(ch.l);
 while(sizeof(ch.q) = 0)
     wait(ch.v, ch.l);
 d := dequeue(ch.q);
 release(ch.l);
 d}
```

Fig. 1. A message passing program synchronized using a monitor

Several approaches to verify termination, deadlock-freedom, liveness, and finite blocking of threads of programs have been presented. Some of these approaches only work with non-blocking algorithms [1–3], where the suspension of one thread cannot lead to the suspension of other threads. These approaches are not applicable for condition variables because suspension of a sender thread in Fig. 1, for example, might cause a receiver thread to be blocked forever. Some other approaches are also presented to verify termination of programs using some blocking constructs such as channels [4–6] and semaphores [7]. These approaches are not general enough to cover condition variables because unlike the channels and semaphores a notification of a condition variable is lost when there is no thread waiting for that condition variable. There are also some studies [8–10] to verify correctness of programs that support condition variables. However, these approaches either only cover a very specific application of condition variables, such as a buffer program with only one producer and one consumer, or are not modular and suffer from a long verification time when the size of the state space, such as the number of threads, is increased.

In this paper we present a modular approach to verify deadlock-freedom of programs in the presence of condition variables. More specifically, this approach makes sure that for any condition variable v for which a thread is waiting there exists a thread obliged to fulfil an obligation for v that only waits for a waitable object whose wait level, an arbitrary number associated with each waitable object, is less than the wait level of v. The presented approach is modular, meaning that different modules (functions) of a program can be verified individually. This approach is based on Leino *et al.* [4] approach for verification of deadlock-freedom in the presence of channels and locks, which in turn was based on Kobayashi's [6] type system for verifying deadlock-freedom of π-calculus processes, and extends the separation logic-based encoding [11] by covering condition variables. We implemented the proposed proof rules in the VeriFast verifier [12–14] and succeeded in verifying some common applications of condition variables such as bounded/unbounded buffer, sleeping barber [15], barrier, and readers-writers locks (see the full version of this paper [16] reporting the verification time of these programs).

This paper is structured as follows. Section 2 provides some background information on the existing approaches upon which we build our verification algorithm. Section 3 introduces a preliminary approach for verifying deadlock-freedom of some common applications of condition variables. In Sect. 4 the precedence relation, aiming to avoid cycles, is relaxed, making it possible to verify some trickier applications of condition variables. A soundness proof of the presented approach is lastly given in Sect. 5.

2 Background Information on the Underlying Approaches

In this section we provide some background information on the existing approaches that verify absence of data races and deadlock in the presence of locks and channels that we build on.

2.1 Verifying Absence of Data Races

Locks/mutexes are mostly used to avoid data races, an undesired situation where a heap location is being written and accessed concurrently by two different threads. One common approach to verify absence of these undesired conditions is ownership: ownership of heap locations is assigned to threads and it is verified that a thread accesses only the heap locations that it owns. Transferring ownership of heap locations between threads is supported through locks by allowing locks, too, to own heap locations. While a lock is not held by a thread, it owns the heap locations described by its *invariant*. More specifically, when a lock is created the resources specified by its invariant are transferred from the creating thread to the lock, when that lock is acquired these resources are transferred from the lock to the acquiring thread, and when that lock is released these resources, that must be again in possession of the thread, are again transferred from the thread to the lock [17]. Figure 2 illustrates how a program increasing a

```
x:=newint(0);
{x↦0}
l := newlock;
{ulock(l) ∗ x↦0}
ct := counter(x:=x, l:=l);
{ulock(ct.l) ∗ ct.x↦0}
{ulock(ct.l) ∗ inv(ct)}
{lock(ct.l) ∧ I(l)=inv(ct)}
{lock(ct.l) ∗ lock(ct.l)}
fork (inc(ct));
{lock(ct.l)}
inc(ct)
```

```
routine inc(counter ct){
 {lock(ct.l) ∧ I(l)=inv(ct)}
 acquire(ct.l);
 {locked(ct.l) ∗ ∃z. ct.x↦z}
 ct.x:=ct.x+1;
 {locked(ct.l) ∗ ∃z. ct.x↦z}
 release(ct.l)
 {lock(ct.l)}}
```

Fig. 2. Verification of data-race-freedom of a program, where $\mathsf{inv} = \lambda ct.\ \exists z.\ ct.x \mapsto z$

counter, which consists of an integer variable x and a lock l protecting this variable, can be verified, where two threads try to write on the variable x. We use separation logic [18] to reason about the ownership of permissions. As indicated below each command, creating the integer variable x initialized by zero provides a read/write access permission to x, denoted by $x \mapsto 0$. This ownership, that is going to be protected by lock l, is transferred to the lock because it is asserted by the lock invariant inv, which is associated with the lock, as denoted by function I, at the point where the lock is initialized. The resulting lock permission, that can be duplicated, is used in the routine inc, where x is increased under protection of lock l. Acquiring this lock in this routine provides a full access permission to x and transforms the lock permission to a locked permission, implying that the related lock has been acquired. Releasing that lock again consumes this access permission and transforms the locked permission to a lock one.

2.2 Verifying Absence of Deadlock

One potential problem with programs using locks and other synchronization mechanisms is deadlock, an undesired situation where all threads of the program are waiting for some waitable objects. For example, a program can deadlock if a thread acquires a lock and forgets to release it, because any other thread waiting for that lock never succeeds in acquiring that lock. As another example, if in a message passing program the number of threads trying to receive a message from a channel is greater than the number of messages sent on that channel there will be some threads waiting for that channel forever. One approach to verify deadlock-freedom of channels and locks is presented by Leino *et al.* [4] that guarantees deadlock-freedom of programs by ensuring that (1) for any *obligee* thread waiting for a waitable object, such as a channel or lock, there is an *obligation* for that object that must be fulfilled by an *obligor* thread, where a thread can fulfil an obligation for a channel/lock if it sends a message on that channel/releases that lock, and (2) each thread waits for an object only if the *wait level* of that object, an arbitrary number assigned to each waitable object, is lower than the wait levels of all obligations of that thread. The second rule is established by making sure that when a thread with some obligations O executes a command $\mathsf{acquire}(o)/\mathsf{receive}(o)$ the precondition $o \prec O$ holds, i.e. the wait level of o is lower than the wait levels of obligations in O. To meet the first rule where the waitable object is a lock, as the example in the left side of Fig. 3 illustrates, after acquiring a lock, that lock is loaded onto the bag[1] (multiset) of obligations of the thread, denoted by $\mathsf{obs}(O)$. This ensures that if a thread tries to acquire a lock that has already been acquired then there is one thread obliged to fulfil an obligation for that lock.

```
{obs(O) ∗ lock(l) ∧ l≺O}
acquire(l);
{obs(O⊎{l}) ∗ locked(l) ∗ I(l)}
...
{obs(O⊎{l}) ∗ locked(l) ∗ I(l)}
release(l)
{obs(O) ∗ lock(l)}
```

```
{obs(O)}
{obs(O⊎{ch}) ∗ credit(ch)}
fork (
 {obs({}) ∗ credit(ch) ∧ ch≺{}}
 receive(ch)
 {obs({})}
);
{obs(O⊎{ch})}
send(ch, 12) {obs(O)}
```

Fig. 3. Verification of deadlock-freedom of locks (left side) and channels (right side)

To establish the first rule where the waitable object is a channel any thread trying to receive a message from a channel ch must spend one *credit* for ch. This credit is normally obtained from the thread that has forked the receiver thread, where this credit is originally created by loading ch onto the bag of obligations of the forking thread. The forking thread can discharge the loaded obligation by either sending a message on the corresponding channel or delegating it to a child thread that can discharge it. The example on the right side of Fig. 3 shows the verification of deadlock-freedom a program in which the main routine, after forking a obligee thread trying to receive a message from channel ch, sends a message on this channel. Before forking the receiver thread, a credit and an obligation for the channel ch are created in the main thread. The former is given to the forked thread, where this credit is spent by the $\mathsf{receive}(ch)$ command, and the latter is fulfilled by the main thread when it executes the command $\mathsf{send}(ch, 12)$.

More formally, the mentioned verification approach satisfies the first rule by ensuring that for each channel ch in the program the number of obligations for ch is equal to/greater than the number of threads waiting for ch. This assurance is obtained by preserving the invariant $Wt(ch)+Ct(ch) \leqslant Ot(ch)+\mathsf{sizeof}(ch)$, while the programming language itself ensures that $\mathsf{sizeof}(ch) > 0 \Rightarrow Wt(ch) = 0$, where sizeof is a function mapping each channel to the size of its queue, $Wt(ch)$

[1] We treat bags of waitable objects as functions from waitable objects to natural numbers.

is the total number of threads currently waiting for channel ch, $Ot(ch)$ is the total number of obligations for channel ch held by all threads, and $Ct(ch)$ is the total number of credits for channel ch currently in the system.

2.3 Proof Rules

The separation logic-based proof rules, introduced by Jacobs *et al.* [11], avoiding data races and deadlock in the presence of locks and channels are shown in Fig. 4, where R and I are functions mapping a waitable object/lock to its wait level/invariant, respectively, and $\mathsf{g\_initl}$, and $\mathsf{g\_load}$ are some *ghost commands* used to initialize an uninitialized lock permission and load a channel onto the bag of obligations and credits of a thread, respectively. When a lock is created, as shown in NewLock, an uninitialized lock permission $\mathsf{ulock}(l)$ is provided for that thread. Additionally, an arbitrary integer number z can be decided as the wait level of that lock that is stored in R. Note that variable z in this rule is universally quantified over the rule, and different applications of the NewLock rule can use different values for this variable. The uninitialized lock permission, as shown in InitLock, can be converted to a normal lock permission $\mathsf{lock}(l)$ provided that the resources described by the invariant of that lock, stored in I, that must be in possession of the thread, are transferred from the thread to the lock. By the rule Acquire, having a lock permission, a thread can acquire that lock if the wait levels of obligations of that thread are all greater than the wait level of that lock. After acquiring the lock, the resources represented by the invariant of that lock are provided for the acquiring thread and the permission lock is converted to a locked permission. When a

$$\textsc{NewLock}\quad \{\mathsf{true}\}\ \mathsf{newlock}\ \{\lambda l.\ \mathsf{ulock}(l) \wedge \mathsf{R}(l){=}z\}$$

$$\textsc{InitLock}\quad \{\mathsf{ulock}(l) * i\}\ \mathsf{g\_initl}(l)\ \{\lambda\_.\ \mathsf{lock}(l) \wedge \mathsf{I}(l){=}i\}$$

$$\textsc{Acquire}\ \{\mathsf{lock}(l) * \mathsf{obs}(O) \wedge l{\prec}O\}\ \mathsf{acquire}(l)\ \{\lambda\_.\ \mathsf{obs}(O{\uplus}\{\!\!\{l\}\!\!\}) * \mathsf{locked}(l) * \mathsf{I}(l)\}$$

$$\textsc{Release}\ \{\mathsf{obs}(O) * \mathsf{locked}(l) * \mathsf{I}(l)\}\ \mathsf{release}(l)\ \{\lambda\_.\ \mathsf{obs}(O{-}\{\!\!\{l\}\!\!\}) * \mathsf{lock}(l)\}$$

$$\textsc{NewChannel}\quad \{\mathsf{true}\}\ \mathsf{newchannel}\ \{\lambda ch.\ \mathsf{R}(ch){=}z\}$$

$$\textsc{Send}\quad \{\mathsf{obs}(O)\}\ \mathsf{send}(ch, v)\ \{\lambda\_.\ \mathsf{obs}(O{-}\{\!\!\{ch\}\!\!\})\}$$

$$\textsc{Receive}\quad \{\mathsf{obs}(O) * \mathsf{credit}(ch) \wedge ch{\prec}O\}\ \mathsf{receive}(ch)\ \{\lambda\_.\ \mathsf{obs}(O)\}$$

$$\textsc{Fork}\quad \frac{\{a * \mathsf{obs}(O)\}\ c\ \{\lambda\_.\ \mathsf{obs}(\{\!\!\{\}\!\!\})\}}{\{a * \mathsf{obs}(O{\uplus}O')\}\ \mathsf{fork}(c)\ \{\lambda\_.\ \mathsf{obs}(O')\}}\qquad \textsc{DupLock}\quad \mathsf{lock}(l) \Leftrightarrow \mathsf{lock}(l) * \mathsf{lock}(l)$$

$$\textsc{LoadOb}\ \{\mathsf{obs}(O)\}\ \mathsf{g\_load}(ch)\ \{\lambda\_.\ \mathsf{obs}(O{\uplus}\{\!\!\{ch\}\!\!\}) * \mathsf{credit}(ch)\}$$

Fig. 4. Proof rules ensuring deadlock-freedom of channels and locks, where $o{\prec}O \Leftrightarrow \forall o' \in O.\ \mathsf{R}(o) < \mathsf{R}(o')$

thread releases a lock, as shown in the rule Release, the resources indicated by the invariant of that lock, that must be in possession of the releasing thread, are transferred from the thread to the lock and the permission locked is again converted to a lock permission. By the rule Receive a thread with obligations O can try to receive a message from a channel ch only if the wait level of ch is lower than the wait levels of all obligations in O. This thread must also spend one credit for ch, ensuring that there is another thread obliged to fulfil an obligation for ch. As shown in the rule Send, an obligation for this channel can be discharged by sending a message on that channel. Alternatively, by the rule Fork, a thread can discharge an obligation for a channel if it delegates that obligation to a child thread, provided that the child thread discharges the delegated obligation. In this setting the verification of a program starts with an empty bag of obligations and must also end with such bag implying that there is no remaining obligation to fulfil.

However, this verification approach is not straightforwardly applicable to condition variables. A command notify cannot be treated like a command send because a notification on a condition variable is lost when there is no thread waiting for that variable. Accordingly, it does not make sense to discharge an obligation for a condition variable whenever it is notified. Similarly, a command wait cannot be treated like a command $\mathsf{receive}$. A command wait is normally executed in a while loop, checking the *waiting condition* of the related condition variable. Accordingly, it is impossible to build a loop invariant for such a loop if we force the wait command to spend a credit for the related condition variable.

3 Deadlock-Free Monitors

3.1 High-Level Idea

In this section we introduce an approach to verify deadlock-freedom of programs in the presence of condition variables. This approach ensures that the verified program never deadlocks, i.e. there is always a running thread, that is not blocked, until the program terminates. The main idea behind this approach is to make sure that for any condition variable v for which a thread is waiting there exists a thread obliged to fulfil an obligation for v that only waits for a waitable object whose wait level is less than the wait level of v. As a consequence, if the program has some threads suspended, waiting for some obligations, there is always a thread obliged to fulfil the obligation o_{min} that is not suspended, where o_{min} has a minimal wait level among all waitable objects for which a thread is waiting. Accordingly, the proposed proof rules make sure that (1) when a command $\mathsf{wait}(v, l)$ is executed $Ot(v) > 0$, where Ot maps each condition variable v to the total number of obligations for v held by all threads (note that having a thread with permission $\mathsf{obs}(O)$ implies $O(v) \leqslant Ot(v)$), (2) a thread discharges an obligation for a condition variable only if after this discharge the invariant $\mathsf{one\_ob}(v, Wt, Ot)$ defined as $Wt(v) > 0 \Rightarrow Ot(v) > 0$ still holds, where $Wt(v)$ denotes the number of threads waiting for condition variable v, and (3) a thread with obligations O executes a command $\mathsf{wait}(v, l)$ only if $v \prec O$.

3.2 Tracking Numbers of Waiting Threads and Obligations

For all condition variables associated with a lock l the value of functions Wt and Ot can only be changed by a thread that has locked l; $Wt(v)$ is changed only when one of the commands $\mathsf{wait}(v, l)$/$\mathsf{notify}(v)$/$\mathsf{notifyAll}(v)$ is executed, requiring holding lock l, and we allow $Ot(v)$ to be changed only when a permission locked for l is available. Accordingly, when a thread acquires a lock these two bags are stored in the related locked permission and are used to establish the rules number 1 and 2, when a thread executes a wait command or discharges one of its obligations. Note that the domain of these functions is the set of the condition variables associated with the related lock. The thread executing the critical section can change these two bags under some circumstances. If that thread loads/discharges a condition variable onto/from the list of its obligations this condition variable must also be loaded/discharged onto/from the bag Ot stored in the related locked permission. Note that unlike the approach presented by Leino *et al.* [4], an obligation for a condition variable can arbitrarily be loaded or discharged by a thread, provided that the rule number 2 is respected. At the start of the execution of a $\mathsf{wait}(v, l)$ command, $Wt(v)$ is incremented and after execution of commands $\mathsf{notify}(v)$/$\mathsf{notifyAll}(v)$ one/all instance(s) of v is/are removed from the bag Wt stored in the related locked permission, since these commands change the number of threads waiting for v.

A program can be successfully verified according to the mentioned rules, formally indicated in Fig. 5, if each lock associated with any condition variable v has an appropriate invariant such that it implies the desired invariant $\mathsf{one\_ob}(v, Wt, Ot)$. Accordingly, the proof rules allow locks to have invariants parametrized over the bags Wt and Ot. When a thread acquires a lock the result of applying the invariant of that lock to these two bags, stored in the related locked permission, is provided for the thread and when that lock is released it is expected that the result of applying the lock invariant to those bags, stored in the related locked permission, again holds. However, before execution of a command $\mathsf{wait}(v, l)$, when lock l with bags Wt and Ot stored in its locked permission is going to be released, it is expected that the invariant of l holds with bags $Wt \uplus \{\!\{v\}\!\}$ and Ot because the running thread is going to wait for v and this condition variable is going to be added to Wt. As this thread resumes its execution, when it has some bags Wt' and Ot' stored in the related locked permission, the result of applying the invariant of l to these bags is provided for that thread. Note that the total number of threads waiting for v, $Wt(v)$, is already decreased when a command $\mathsf{notify}(v)$ or $\mathsf{notifyAll}(v)$ is executed, causing the waiting thread(s) to wake up and try to acquire the lock associated with v.

3.3 Resource Transfer on Notification

In general, as we will see when looking at examples, it is sometimes necessary to transfer resources from a notifying thread to the threads being notified[2].

[2] This transfer is only sound in the absence of spurious wake-ups, where a thread is awoken from its waiting state even though no thread has signaled the related condition variable.

To this end, these resources, specified by a function M, are associated with each condition variable v when v is created, such that the commands $\mathsf{notify}(v)$/$\mathsf{notifyAll}(v)$ consume one/$Wt(v)$ instance(s) of these resources, respectively, and the command $\mathsf{wait}(v, l)$ produces one instance of such resources (see the rules Wait,Notify, and NotifyAll in Fig. 5).

$$\textsc{NewLock}\ \{\mathsf{true}\}\ \mathsf{newlock}\ \{\lambda l.\ \mathsf{ulock}(l, \{\!\!\}, \{\!\!\}) \land \mathsf{R}(l){=}z\}$$

$$\textsc{NewCv}\ \{\mathsf{true}\}\ \mathsf{newcond}\ \{\lambda v.\ \mathsf{R}(v){=}z \land \mathsf{L}(v){=}l \land \mathsf{M}(v){=}m\}$$

$$\textsc{Acquire}\ \begin{array}{c}\{\mathsf{lock}(l) * \mathsf{obs}(O) \land l{\prec}O\}\ \mathsf{acquire}(l)\\ \{\lambda\_.\ \exists\, Wt, Ot.\ \mathsf{locked}(l, Wt, Ot) * \mathsf{I}(l)(Wt, Ot) * \mathsf{obs}(O{\uplus}\{\!\!\{l\}\!\!\})\}\end{array}$$

$$\begin{array}{l}\textsc{Release}\\ \{\mathsf{locked}(l, Wt, Ot) * \mathsf{I}(l)(Wt, Ot) * \mathsf{obs}(O{\uplus}\{\!\!\{l\}\!\!\})\}\ \mathsf{release}(l)\ \{\lambda\_.\ \mathsf{lock}(l) * \mathsf{obs}(O)\}\end{array}$$

$$\textsc{Wait}\quad \begin{array}{c}\{\mathsf{locked}(l, Wt, Ot) * \mathsf{I}(l)(Wt{\uplus}\{\!\!\{v\}\!\!\}, Ot) * \mathsf{obs}(O{\uplus}\{\!\!\{l\}\!\!\})\\ \land\ l{=}\mathsf{L}(v) \land v{\prec}O \land l{\prec}O \land \mathsf{safe\_obs}(v, Wt{\uplus}\{\!\!\{v\}\!\!\}, Ot)\}\ \mathsf{wait}(v, l)\\ \{\lambda\_.\ \mathsf{obs}(O{\uplus}\{\!\!\{l\}\!\!\}) * \exists\, Wt', Ot'.\ \mathsf{locked}(l, Wt', Ot') * \mathsf{I}(l)(Wt', Ot') * \mathsf{M}(v)\}\end{array}$$

$$\textsc{Notify}\ \begin{array}{c}\{\mathsf{locked}(\mathsf{L}(v), Wt, Ot) * (Wt(v) = 0 \lor \mathsf{M}(v))\}\ \mathsf{notify}(v)\\ \{\lambda\_.\ \mathsf{locked}(\mathsf{L}(v), Wt{-}\{\!\!\{v\}\!\!\}, Ot)\}\end{array}$$

$$\begin{array}{l}\textsc{NotifyAll}\\ \{\mathsf{locked}(\mathsf{L}(v), Wt, Ot) * (\overset{Wt(v)}{\underset{i:=0}{*}} \mathsf{M}(v))\}\ \mathsf{notifyAll}(v)\ \{\lambda\_.\ \mathsf{locked}(\mathsf{L}(v), Wt[v{:=}0], Ot)\}\end{array}$$

$$\begin{array}{l}\textsc{InitLock}\\ \{\mathsf{ulock}(l, Wt, Ot) * inv(Wt, Ot) * \mathsf{obs}(O)\}\ \mathsf{g\_initl}(l)\ \{\lambda\_.\ \mathsf{lock}(l) * \mathsf{obs}(O) \land \mathsf{I}(l){=}inv\}\end{array}$$

$$\textsc{ChargeOb}\ \begin{array}{c}\{\mathsf{obs}(O) * \mathsf{ulock/locked}(\mathsf{L}(v), Wt, Ot)\}\ \mathsf{g\_chrg}(v)\\ \{\lambda\_.\ \mathsf{obs}(O{\uplus}\{\!\!\{v\}\!\!\}) * \mathsf{ulock/locked}(\mathsf{L}(v), Wt, Ot{\uplus}\{\!\!\{v\}\!\!\})\}\end{array}$$

$$\textsc{DisOb}\ \begin{array}{c}\{\mathsf{obs}(O) * \mathsf{ulock/locked}(\mathsf{L}(v), Wt, Ot) \land \mathsf{safe\_obs}(v, Wt(v), Ot{-}\{\!\!\{v\}\!\!\})\}\\ \mathsf{g\_disch}(v)\ \{\lambda\_.\ \mathsf{obs}(O{-}\{\!\!\{v\}\!\!\}) * \mathsf{ulock/locked}(\mathsf{L}(v), Wt, Ot{-}\{\!\!\{v\}\!\!\})\}\end{array}$$

Fig. 5. Proof rules to verify deadlock-freedom of condition variables, where $Wt(v)$ and $Ot(v)$ denote the total number of threads waiting for v and the total number of obligations for v, respectively, and $\mathsf{safe\_obs}(v, Wt, Ot) \Leftrightarrow \mathsf{one\_ob}(v, Wt, Ot)$ and $\mathsf{one\_ob}(v, Wt, Ot) \Leftrightarrow (Wt(v) > 0 \Rightarrow Ot(v) > 0)$

3.4 Proof Rules

Figure 5 shows the proposed proof rules used to verify deadlock-freedom of condition variables, where L and M are functions mapping each condition variable to its associated lock and to the resources that are moved from the notifying thread to the notified one when that condition variable is notified, respectively.

Creating a lock, as shown in the rule NewLock, produces a permission ulock storing the bags Wt and Ot, where these bags are initially empty. The bag Ot in this permission, similar to a locked one, can be changed provided that the obligations of the running thread are also updated by one of the ghost commands $\mathsf{g\_chrg}(v)$ or $\mathsf{g\_disch}(v)$ (see rules ChargeOb and DisOb). The lock related to this permission can be initialized by transferring the resources described by the invariant of this lock, that is now parametrized over the bags Wt and Ot, applied to the bags stored in this permission from the thread to the lock (see rule InitLock). When this lock is acquired, as shown in the rule Acquire, the resources indicated by its invariant are provided for the thread, and when it is released, as shown in the rule Release, the resources described by its invariant that must hold with appropriate bags, are again transferred from the thread to the lock. The rules Wait and DisOb ensure that for any condition variable v when the number of waiting threads is increased, by executing a command $\mathsf{wait}(v, l)$, or the number of the obligations is decreased, by (logically) executing a command $\mathsf{g\_disch}(v)$, the desired invariant one_ob still holds. Additionally, the rules Acquire and Wait make sure that a thread only waits for a waitable object whose wait level is lower that the wait levels of obligations of that thread. Note that in the rule Wait in the precondition of the command $\mathsf{wait}(v, l)$ it is not necessary that the wait level of v is lower that the wait level of l, since lock l is going to be released by this command. However, in this precondition the wait level of l must be lower that the wait levels of the obligations of the thread because when this thread is notified it tries to reacquire l, at which point $l \prec O$ must hold. The commands $\mathsf{notify}(v)$/$\mathsf{notifyAll}(v)$, as shown in the rules Notify and NotifyAll, remove one/all instance(s) of v, if any, from the bag Wt stored in the related locked permission. Additionally, $\mathsf{notify}(v)$ consumes the moving resources, indicated by $\mathsf{M}(v)$, that appear in the postcondition of the notified thread. Note that $\mathsf{notifyAll}(v)$ consumes $Wt(v)$ instances of these resources, since they are transferred to $Wt(v)$ threads waiting for v.

3.5 Verifying Channels

Ghost Counters. We will now use our proof system to prove deadlock-freedom of the program in Fig. 1. To do so, however, we will introduce a *ghost resource* that plays the role of *credits*, in such a way that we can prove the invariant $Wt(ch) + Ct(ch) \leqslant Ot(ch) + \mathsf{sizeof}(ch)$. In particular, we want this property to follow from the lock invariant. This means we need to be able to talk, in the lock invariant, about the total number of credits in the system. To achieve this, we introduce a notion of *ghost counters* and corresponding *ghost counter tickets*, both of which are a particular kind of ghost resources. Specifically, we introduce three ghost commands: g_newctr, g_inc, and g_dec. g_newctr allocates a new ghost counter whose *value* is zero and returns a *ghost counter identifier* c for it. $\mathsf{g\_inc}(c)$ increments the value of the ghost counter with identifier c and produces a *ticket* for the counter. $\mathsf{g\_dec}(c)$, finally, consumes a ticket for ghost

$$\textsc{NewCounter}\ \{\mathsf{true}\}\ \mathsf{g\_newctr}\ \{\lambda c.\ \mathsf{ctr}(c, 0)\}$$

$$\textsc{IncCounter}\ \{\mathsf{ctr}(c, n)\}\ \mathsf{g\_inc}(c)\ \{\lambda\_.\ \mathsf{ctr}(c, n{+}1) * \mathsf{tic}(c)\}$$

$$\textsc{DecCounter}\ \{\mathsf{ctr}(c, n) * \mathsf{tic}(c)\}\ \mathsf{g\_dec}(c)\ \{\lambda\_.\ \mathsf{ctr}(c, n{-}1) \wedge 0{<}n\}$$

Fig. 6. Ghost counters

counter c and decrements the ghost counter's value. Since these are the only operations that manipulate ghost counters or ghost counter tickets, it follows that the value of a ghost counter c is always equal to the number of tickets for c in the system. Proof rules for these ghost commands are shown in Fig. 6[3].

The Channels Proof. Figure 7 illustrates how the program in Fig. 1 can be verified using our proof system. The invariant of lock $ch.l$ in this program, denoted by $\mathsf{inv}(ch)$, is parametrized over bags Wt, Ot and implies the desired invariant $\mathsf{one\_ob}(ch.v,\ Wt, Ot)$. The permission $\mathsf{ctr}(ch.c, Ctv)$ in this invariant indicates that the total number of credits (tickets) for $ch.v$ is Ctv, where $ch.c$ is a *ghost field* added to the channel data structure, aiming to store a ghost counter identifier for the ghost counter of $ch.v$. Generally, a lock invariant can imply the invariant $\mathsf{one\_ob}(v,\ Wt, Ot)$ if it asserts $Wt(v) + Ct(v) \leqslant Ot(v) + S(v)$ and $Wt(v) \leqslant Ot(v)$, where $Ct(v)$ is the total number of credits for v and $S(v)$ is an integer value such that the command $\mathsf{wait}(v, l)$ is executed only if $S(v) \leqslant 0$. After initializing l in the main routine, there exists a credit for $ch.v$ (denoted by $\mathsf{tic}(ch.c)$) that is consumed by the thread executing the receive routine, and also an obligation for $ch.v$ that is fulfilled by this thread after executing the send routine. The credit $\mathsf{tic}(ch.c)$ in the precondition of the routine receive ensures that before execution of the command $\mathsf{wait}(ch.v, ch.l)$, $Ot(ch.v) > 0$. This inequality follows from the invariant of lock l, which holds for $Wt \uplus \{\!\{ch.v\}\!\}$ and Ot when Ctv is decreased by $\mathsf{g\_dec}(ch.c)$. This credit (or the one specified by $\mathsf{M}(ch.v)$ that is moved from a notifier thread when the receiver thread wakes up) must be consumed after execution of the command $\mathsf{dequeue}(ch.q)$ and before releasing $ch.l$ to make sure that the invariant still holds after decreasing the number of items in $ch.q$. The obligation for $ch.v$ in the precondition of the routine send is discharged by this routine, which is safe, since after the execution of the commands enqueue and notify the invariant $\mathsf{one\_ob}(ch.v,\ Wt, Ot - \{\!\{ch.v\}\!\})$, which follows from the lock invariant, holds.

[3] Some logics for program verification, such as Iris [19], include general support for defining ghost resources such as our ghost counters. In particular, our ghost counters can be obtained in Iris as an instance of the *authoritative monoid* [19, p. 5].

```
inv(channel $ch$) ::= $\lambda Wt.\ \lambda Ot.\ \exists Ctv.$ ctr($ch.c, Ctv$) $*$ $\exists s.$ queue($ch.q, s$) $\wedge$
 L($ch.v$)=$ch.l$ $\wedge$ M($ch.v$)=tic($ch.c$) $\wedge$
 $Wt(ch.v) + Ctv \leqslant Ot(ch.v) + s \wedge$
 $Wt(ch.v) \leqslant Ot(ch.v)$

routine main(){{obs({})}
 $q$:=newqueue; $l$:=newlock; $v$:=newcond; $c$:=g_newctr; g_inc($c$);
 {obs({}) $*$ ulock($l$, {}, {}) $*$ queue($q$, 0) $*$ ctr($c$, 1) $*$ tic($c$)
 $\wedge$ L($v$)=$l$ $\wedge$ M($v$)=tic($c$) $\wedge$ R($l$)=0 $\wedge$ R($v$)=1}
 $ch$:=channel($q, l, v$); $ch.c$:=$c$;
 {obs({}) $*$ ulock($l$, {}, {}) $*$ inv($ch$)({}, {$v$}) $*$ tic($c$)} g_chrg($v$);
 {obs({$v$}) $*$ ulock($l$, {}, {$v$}) $*$ inv($ch$)({}, {$v$}) $*$ tic($c$)} g_initl($l$);
 {obs({$v$}) $*$ lock($l$) $*$ tic($c$) $\wedge$ I($l$)=inv($ch$)}
 fork (receive($ch$));
 {obs({$v$}) $*$ lock($l$)}
 send($ch$, 12) {obs({})}}

routine receive(channel $ch$){
 {obs($O$) $*$ tic($ch.c$) $*$ lock($ch.l$) $\wedge$ $ch.l \prec O$ $\wedge$ $ch.v \prec O$ $\wedge$ I($ch.l$)=inv($ch$)}
 acquire($ch.l$);
 {obs($O \uplus$ {$ch.l$}) $*$ tic($ch.c$) $*$ $\exists Wt, Ot.$ locked($ch.l$, $Wt, Ot$) $*$ inv($ch$)($Wt, Ot$)}
 while(sizeof($ch.q$) = 0){ g_dec($ch.c$);
   {obs($O \uplus$ {$ch.l$}) $*$ $\exists Wt, Ot.$ locked($ch.l$, $Wt, Ot$) $*$ inv($ch$)($Wt \uplus$ {$ch.v$}, $Ot$)}}
   wait($ch.v, ch.l$)
   {obs($O \uplus$ {$ch.l$}) $*$ M($ch.v$) $*$ $\exists Wt, Ot.$ locked($ch.l$, $Wt, Ot$) $*$ inv($ch$)($Wt, Ot$)}};
 dequeue($ch.q$); g_dec($ch.c$);
 {obs($O \uplus$ {$ch.l$}) $*$ $\exists Wt, Ot.$ locked($ch.l$, $Wt, Ot$) $*$ inv($ch$)($Wt, Ot$)}
 release($ch.l$) {obs($O$) $*$ lock($ch.l$)}}

routine send(channel $ch$, int $d$){
 {obs($O \uplus$ {$ch.v$}) $*$ lock($ch.l$) $\wedge$ $ch.l \prec O \uplus$ {$ch.v$} $\wedge$ I($ch.l$)=inv($ch$)}
 acquire($ch.l$);
 {obs($O \uplus$ {$ch.v, ch.l$}) $*$ $\exists Wt, Ot.$ locked($ch.l$, $Wt, Ot$) $*$ inv($ch$)($Wt, Ot$)}
 enqueue($ch.q, d$);
 if ($Wt(ch.v)$>0) g_inc($ch.c$);
 notify($ch.v$);
 {obs($O \uplus$ {$ch.v, ch.l$}) $*$ $\exists Wt, Ot.$ locked($ch.l$, $Wt, Ot$) $*$ inv($ch$)($Wt, Ot -$ {$ch.v$})}
 g_disch($ch.v$);
 {obs($O \uplus$ {$ch.l$}) $*$ $\exists Wt, Ot.$ locked($ch.l$, $Wt, Ot$) $*$ inv($ch$)($Wt, Ot$)}
 release($ch.l$) {obs($O$) $*$ lock($ch.l$)}}
```

Fig. 7. Verification of the program in Fig. 1

3.6 Other Examples

Using the proof system of this section we prove two other deadlock-free programs, namely *sleeping barber* [16], and *barrier*. In the barrier program shown in Fig. 8, a barrier b consists of an integer variable r indicating the number of the remaining

```
routine main(){
 $r$:=newint(3);
 $l$:=newlock;
 $v$:=newcond;
 $b$:=barrier($r, l, v$);
 fork (task$_1$(); wait_for_rest($b$); task$_2$());
 fork (task$_1$(); wait_for_rest($b$); task$_2$());
 task$_1$(); wait_for_rest($b$); task$_2$()}
```

```
routine wait_for_rest(barrier $b$){
 acquire($b.l$);
 $b.r$:=$b.r-1$;
 if($b.r=0$)
   notifyAll();
 else
   while($b.r>0$)
      wait($b.v, b.l$);
 release($b.l$)}
```

$$\mathsf{inv}(\mathsf{barrier}\ b) ::= \lambda Wt.\ \lambda Ot.\ \exists r \geqslant 0.\ b.r \mapsto r \wedge \mathsf{L}(b.v)=b.l \wedge \mathsf{M}(b.v)=\mathsf{true} \wedge \\ (Wt(b.v) = 0 \vee 0 < r)\ \wedge\ (r \leqslant Ot(b.v))$$

```
routine main(){$\{\mathsf{obs}(\{\!\})\}$
 $r$:=newint(3); $l$:=newlock; $v$:=newcond;
 $\{\mathsf{obs}(\{\!\}) * r \mapsto 3 * \mathsf{ulock}(l, \{\!\}, \{\!\}) \wedge \mathsf{L}(v)=l \wedge \mathsf{M}(v)=\mathsf{true} \wedge \mathsf{R}(l)=0 \wedge \mathsf{R}(v)=1\}$
 $b$:=barrier($r, l, v$);
 $\{\mathsf{obs}(\{\!\}) * \mathsf{inv}(b)(\{\!\}, \{3 \cdot v\}) * \mathsf{ulock}(l, \{\!\}, \{\!\})\}$
 g_chrg($v$); g_chrg($v$); g_chrg($v$); g_initl($l$);
 $\{\mathsf{obs}(\{3 \cdot v\}) * \mathsf{lock}(l) \wedge \mathsf{I}(l)=\mathsf{inv}(b)\}$
 fork (wait_for_rest($b$));
 $\{\mathsf{obs}(\{2 \cdot v\}) * \mathsf{lock}(l)\}$
 fork (wait_for_rest($b$));
 $\{\mathsf{obs}(\{v\}) * \mathsf{lock}(l)\}$
 wait_for_rest($b$) $\{\mathsf{obs}(\{\!\})\}$}
```

```
routine wait_for_rest(barrier $b$){
 $\{\mathsf{obs}(O \uplus \{b.v\}) * \mathsf{lock}(b.l) \wedge b.l \prec O \uplus \{b.v\} \wedge b.v \prec O \wedge \mathsf{I}(b.l)=\mathsf{inv}(b)\}$
 acquire($b.l$);
 $\{\mathsf{obs}(O \uplus \{b.v, b.l\}) * \exists Wt, Ot.\ \mathsf{locked}(b.l,\ Wt, Ot) * \mathsf{inv}(b)(Wt, Ot)\}$
 $b.r$:=$b.r-1$;
 if($b.r=0$){
   notifyAll($b.v$);
   $\{\mathsf{obs}(O \uplus \{b.v, b.l\}) * \exists Wt, Ot.\ \mathsf{locked}(b.l,\ Wt[b.v{:=}0], Ot)$
   $* \mathsf{inv}(b)(Wt[b.v{:=}0], Ot - \{b.v\})\}$ g_disch($b.v$)
   $\{\mathsf{obs}(O \uplus \{b.l\}) * \exists Wt, Ot.\ \mathsf{locked}(b.l,\ Wt, Ot) * \mathsf{inv}(b)(Wt, Ot)\}$}
 else{
   $\{\mathsf{obs}(O \uplus \{b.v, b.l\}) * \exists Wt, Ot.\ \mathsf{locked}(b.l,\ Wt, Ot)$
   $* \mathsf{inv}(b)(Wt, Ot - \{b.v\})\}$ g_disch($b.v$);
   $\{\mathsf{obs}(O \uplus \{b.l\}) * \exists Wt, Ot.\ \mathsf{locked}(b.l,\ Wt, Ot) * \mathsf{inv}(b)(Wt, Ot)\}$
   while($b.r>0$)
      $\{\mathsf{obs}(O \uplus \{b.l\}) * \exists Wt, Ot.\ \mathsf{locked}(b.l,\ Wt, Ot) * \mathsf{inv}(b)(Wt \uplus \{b.v\}, Ot)\}$
      wait($b.v, b.l$)
      $\{\mathsf{obs}(O \uplus \{b.l\}) * \exists Wt, Ot.\ \mathsf{locked}(b.l,\ Wt, Ot) * \mathsf{inv}(b)(Wt, Ot)\}$};
 release($b.l$) $\{\mathsf{obs}(O) * \mathsf{lock}(b.l)\}$}
```

Fig. 8. Verification of a barrier synchronized using a monitor

threads that must call the routine wait_for_rest, a lock l protecting r against data races, and a condition variable v. Each thread executing the routine wait_for_rest first decreases the variable r, and if the resulting value is still positive waits for v, otherwise it notifies all threads waiting for v. In this program the barrier is initialized to 3, implying that no thread must start task_2 unless all the three threads in this program finish task_1. This program is deadlock-free because the routine wait_for_rest is executed by three different threads. Figure 8 illustrates how this program can be verified by the presented proof rules. Note that before executing g_disch in the else branch, safe_obs holds because at this point we have $0 < b.r$, which implies $1 < b.r$ before the execution of $b.r := b.r - 1$, and by the invariant we have $1 < Ot(b.v)$, implying $0 < (Ot - \{\!\{b.v\}\!\})(b.v)$. The interesting point about the verification of this program is that since all the threads waiting for condition variable v in this program are notified by the command notifyAll, the invariant of the related lock, implying one_ob$(b.v, Wt, Ot)$, is significantly different from the ones defined in the channel and sleeping barber examples. Generally, for a condition variable v on which only notifyAll is executed (and not notify) a lock invariant can imply the invariant one_ob(v, Wt, Ot) if it asserts $Wt(v) = 0 \vee S(v) \leqslant Ct(v)$ and $Ct(v) < Ot(v) + S(v)$, where $Ct(v)$ is the total number of credits for v and $S(v)$ is an integer value such that the command wait(v, l) is executed only if $S(v) \leqslant 0$. For this particular example $S(b.v) = 1 - b.r$ and $Ct(b.v) = 0$, since this program can be verified without incorporating the notion of credits.

4 Relaxing the Precedence Relation

The precedence relation, in this paper denoted by $\prec$, introduced in [4] makes sure that all threads wait for the waitable objects in strict ascending order (with respect to the wait level associated with each waitable object), or here in this paper in descending order, ensuring that in any state of the execution there is no cycle in the corresponding wait-for graph. However, this relation is too restrictive and prevents verifying some programs that are actually deadlock-free, such as the one shown in the left side of Fig. 9. In this program a value is increased by two threads communicating through a channel. Each thread receives a value from the channel, increases that value, and then sends it back on the channel. Since an initial value is sent on the related channel this program is deadlock-free. The first attempt to verify this program is illustrated in the middle part of Fig. 9, where the required credit to verify the receive command in the routine inc is going to be provided by the send command, executed immediately after this command, and not by the precondition of this routine. In other words, the idea is to load a credit and an obligation for ch in the routine inc itself, and then spend the loaded credit to verify the receive(ch) command and fulfil the loaded obligation by the send(ch) command. However, this idea fails because the receive command in the routine inc cannot be verified since one of its preconditions, $ch \prec \{\!\{ch\}\!\}$, never holds. Kobayashi [6,20] has addressed this problem in his type system by using the notion of *usages* and assigning levels to each *obligation/capability*, instead of

```
routine main(){
  ch:=channel;
  send(ch, 12);
  fork (inc(ch));
  fork (inc(ch))}

routine inc(channel ch){
 d:=receive(ch);
 send(ch, d+1)}
```

```
routine main(){
 $\{obs(\{\!|\,|\!\})\}$
 ch:=newchannel;
 send(ch, 12);
 fork (inc(ch));
 fork (inc(ch)) $\{obs(\{\!|\,|\!\})\}$}

routine inc(channel ch){
 $\{obs(\{\!|\,|\!\})\}$
 $\{obs(\{\!|ch|\!\}) * credit(ch)$
 $\wedge\ ch \not\prec \{\!|ch|\!\}\}$
 d:=receive(ch);
 $\{obs(\{\!|ch|\!\})\}$
 send(ch, d+1) $\{obs(\{\!|\,|\!\})\}$}
```

```
routine main(){
 $\{obs(\{\!|\,|\!\})\}$
 ch:=newchannel;
 $\{obs(\{\!|ch|\!\}) \wedge P(ch)=true\}$
 send(ch, 12);
 $\{obs(\{\!|\,|\!\})\}$
 fork (inc(ch));
 fork (inc(ch)) $\{obs(\{\!|\,|\!\})\}$}

routine inc(channel ch){
 $\{obs(\{\!|\,|\!\}) \wedge ch \preccurlyeq \{\!|ch|\!\}\}$
 $\langle obs(\{\!|ch|\!\}) * credit(ch)$
 $\wedge\ ch \preccurlyeq \{\!|ch|\!\}\rangle$
 d:=receive(ch);
 $\{obs(\{\!|ch|\!\})\}$
 send(ch, d+1) $\{obs(\{\!|\,|\!\})\}$}
```

Fig. 9. A deadlock-free program verified by exploiting the relaxed precedence relation

waitable objects. However, in the next section we provide a novel idea to address this problem by just relaxing the precedence relation used in the presented proof rules.

4.1 A Relaxed Precedence Relation

To tackle the problem mentioned in the previous section we relax the precedence relation, enforced by $\prec$, by replacing $\prec$ by $\preccurlyeq$ satisfying the following property: $o \preccurlyeq O$ holds if either $o \prec O$ or (1) $o \prec O - \{\!|o|\!\}$, and (2) o satisfies the property that in any execution state, if a thread waits for o then there exists a thread that can discharge an obligation for o and is not waiting for any object whose wait level is equal to/greater than the wait level of o. This property still guarantees that in any state of the execution if the program has some threads suspended, waiting for some obligations, there is always a thread obliged to fulfil the obligation o_{min} that is not blocked, where o_{min} has a minimal wait level among all waitable objects for which a thread is waiting.

The condition number 2 is met if it is an invariant that for a condition variable o for which a thread is waiting the total number of obligations is greater than the total number of waiting threads. Since each thread waiting for o has at most one instance of o in the bag of its obligations, according to the *pigeonhole principle*, if the number obligations for o is higher than the number of threads waiting for o then there exists a thread that holds an obligation for o that is not waiting for o, implying the rule number 2 because this thread only waits for objects whose wait levels are lower than the wait level of o. Accordingly, we first introduce a new function P in the proof rules mapping each waitable object to a boolean value, and then make sure that for any object o for which a thread is waiting if

$\mathsf{P}(o) = \mathsf{true}$ then $Wt(o) < Ot(o)$. With the help of this function we define the relaxed precedence relation as shown in Definition 1.

Definition 1 (Relaxed precedence relation). *The relaxed precedence relation indexed over functions* R *and* P *holds for a waitable object* v *and a bag of obligations* O*, denoted by* $v \preccurlyeq O$*, if and only if:*

$$v \prec O \ \vee \ (v \prec O - \{\!\!\{v\}\!\!\} \wedge \mathsf{P}(v) = \mathsf{true})\ ,\ \textit{where}\ v \prec O \Leftrightarrow \forall o \in O.\ \mathsf{R}(v) < \mathsf{R}(o)$$

Using this relaxed precedence relation the approach presented by Leino *et al.* [4] can also support more complex programs, such as the one in the left side of Fig. 9. This approach can exploit this relation by (1) replacing the original precedence relation $\prec$ by the relaxed one $\preccurlyeq$, and (2) replacing the rule associated with creating a channel by the one shown below. According to this proof rule for each channel ch the function P, in the definition of the relaxed precedence relation, is initialized when ch is created such that if $\mathsf{P}(ch)$ is decided to be true then one obligation for ch is loaded onto the bag of obligations of the creating thread. The approach is still sound because for any channel ch for which P is true the invariant $Wt(ch) + Ct(ch) < Ot(ch) + \mathsf{sizeof}(ch)$ holds. Combined with the fact that in this language, where channels are primitive constructs, $Wt(ch) > 0 \Rightarrow \mathsf{sizeof}(ch) = 0$, we have $Wt(ch) > 0 \Rightarrow Wt(ch) < Ot(ch)$. Now consider a deadlocked state, where each thread is waiting for a waitable object. Among all of these waitable objects take the one having a minimal wait level, namely o_m. If o_m is a lock or a channel, where $\mathsf{P}(o_m) = \mathsf{false}$, then at least one thread has an obligation for o_m and is waiting for an object o whose wait level is lower that the wait level of o_m, which contradicts minimality of the wait level of o_m. Otherwise, since $Wt(o_m) > 0$ we have $Wt(o_m) < Ot(o_m)$. Additionally, we know that each thread waiting for o_m has at most one obligation for o_m. Accordingly, there must be a thread holding an obligation for o_m that is not waiting for o_m. Consequently, this thread must be waiting for an object o whose wait level is lower than the wait level of o_m, which contradicts minimality of the wait level of o_m.

$$\{\mathsf{obs}(O)\}\ \mathsf{newchannel}\ \{\lambda ch.\ \mathsf{obs}(O') \wedge \mathsf{R}(ch) = z \wedge \mathsf{P}(ch) = b \\ \wedge((b = \mathsf{false} \wedge O' = O) \vee (b = \mathsf{true} \wedge O' = O \uplus \{\!\!\{ch\}\!\!\}))\}$$

To exploit the relaxed definition in the approach presented in this paper we only need to make sure that for any condition variable v for which a thread is waiting if $\mathsf{P}(v)$ is true then $Ot(v)$ is greater than $Wt(v)$. To achieve this goal we include this invariant in the definition of the invariant $\mathsf{safe\_obs}$, shown in Definition 2, an invariant that must hold when a command wait or a ghost command $\mathsf{g\_disch}$ is executed.

Definition 2 (Safe Obligations). *The relation* $\mathsf{safe\_obs}(v, Wt, Ot)$*, indexed over function* P*, holds if and only if:*

$$\mathsf{one\_ob}(v, Wt, Ot)\ \wedge\ (\mathsf{P}(v) = \mathsf{true} \Rightarrow \mathsf{spare\_ob}(v, Wt, Ot)),\ \textit{where}$$
$$\mathsf{one\_ob}(v, Wt, Ot) \Leftrightarrow (Wt(v) > 0 \Rightarrow Ot(v) > 0)$$
$$\mathsf{spare\_ob}(v, Wt, Ot) \Leftrightarrow (Wt(v) > 0 \Rightarrow Wt(v) < Ot(v))$$

$$\begin{array}{l}\mathsf{one\_ob}(v,\ Wt, Ot)\ \wedge\ (\mathsf{P}(v){=}\mathsf{true} \Rightarrow \mathsf{spare\_ob}(v,\ Wt, Ot)),\ \textit{where}\\ \mathsf{one\_ob}(v,\ Wt, Ot) \Leftrightarrow (\,Wt(v){>}0 \Rightarrow Ot(v){>}0)\\ \mathsf{spare\_ob}(v,\ Wt, Ot) \Leftrightarrow (\,Wt(v){>}0 \Rightarrow\ Wt(v){<}Ot(v))\end{array}$$

```
routine main(){
 $aw$:=newint(0);
 $ww$:=newint(0);
 $ar$:=newint(0);
 $l$:=newlock;
 $v_w$:=newcond;
 $v_r$:=newcond;
 $b$ := rdwr($aw, ww$
  $, ar, l, v_w, v_r$);
 fork(
  while (true)
   fork(reader($b$))
 );
 while (true)
  fork(writer($b$))
}
```

```
routine reader(rdwr $b$){
 acquire($b.l$);
 while($b.aw$+$b.ww$>0)
   wait($b.v_r, b.l$);
 $b.ar$:=$b.ar$+1;
 release($b.l$);
 // Perform reading ...
 acquire($b.l$);
 if($b.ar$<1)
   abort;
 $b.ar$:=$b.ar$−1;
 notify($b.v_w$);
 release($b.l$)}
```

```
routine writer(rdwr $b$){
 acquire($b.l$);
 while($b.aw$+$b.ar$>0){
   $b.ww$:=$b.ww$+1;
   wait($b.v_w, b.l$);
   if($b.ww$<1)
      abort();
   $b.ww$:=$b.ww$−1
 };
 $b.aw$:=$b.aw$+1;
 release($b.l$);
 // Perform writing ...
 acquire($b.l$);
 if($b.aw$≠1)
   abort;
 $b.aw$:=$b.aw$−1;
 notify($b.v_w$);
 if($b.ww$=0)
   notifyAll($b.v_r$);
 release($b.l$)}
```

Fig. 10. A readers-writers program with variables aw, holding the number of threads writing, ww, holding the number of thread waiting to write, and ar, holding the number of threads reading, that is synchronized using a monitor consisting of condition variables v_w, preventing writers from writing while other threads are reading or writing, and v_r, preventing readers from reading while there is another thread writing or waiting to write.

Readers-Writes Locks. As another application of this relaxed definition consider a readers-writers program, shown in Fig. 10[4], where the condition variable v_w prevents writers from writing on a shared memory when that memory is being accessed by other threads. After reading the shared memory, a reader thread notifies this condition variable if there is no other thread reading that memory. This condition variable is also notified by a writer thread when it finishes its writing. Consequently, a writer thread first might wait for v_w and then fulfil an obligation for this condition variable. This program is verified if the writer thread itself produces a credit and an obligation for v_w and then uses the former for the command wait(v_w, l) and fulfils the latter at the end of its execution. Accordingly, since when the command wait(v_w, l) is executed v_w is in the bag of obligations of the

[4] The abort commands in this program can be eliminated using the ghost counters from Fig. 6. However, we leave them in for simplicity.

```
$\mathsf{inv}(\mathsf{rdwr}\ b) ::= \lambda Wt.\ \lambda Ot.\ \exists Ctw.\ \mathsf{ctr}(b.c_w, Ctw) *$
 $\exists aw \geqslant 0, ww \geqslant 0, ar \geqslant 0.\ b.aw \mapsto aw * b.ww \mapsto ww * b.ar \mapsto ar \wedge$
 $\mathsf{L}(b.v_w) = \mathsf{L}(b.v_r) = b.l \wedge \mathsf{M}(b.v_w) = \mathsf{tic}(b.c_w) \wedge \mathsf{M}(b.v_r) = \mathsf{true} \wedge \mathsf{P}(v_w) = \mathsf{true} \wedge \mathsf{P}(v_r) = \mathsf{false} \wedge$
 $(Wt(b.v_r) = 0 \vee 0 < aw + ww) \wedge$
 $aw + ww \leqslant Ot(b.v_r) \wedge$
 $Wt(b.v_w) + Ctw + aw + ar \leqslant Ot(b.v_w) \wedge$
 $(Wt(b.v_w) = 0 \vee Wt(b.v_w) < Ot(b.v_w))$
```

```
routine main(){
 $aw$:=newint(0); $ww$:=newint(0);
 $ar$:=newint(0); $l$:=newlock;
 $v_w$:=newcond; $v_r$:=newcond;
 $b$ := rdwr($aw, ww, ar, l, v_w, v_r$);
 $b.c_w$:=g_newctr;
 {obs({}) * inv($b$)({}, {}) * ulock($l$, {}, {}) *
 $\mathsf{L}(v_w) = \mathsf{L}(v_r) = l \wedge \mathsf{M}(v_w) = \mathsf{tic}(b.c_w) \wedge$
 $\mathsf{M}(v_r) = \mathsf{true} \wedge \mathsf{R}(l) = 0 \wedge \mathsf{R}(v_w) = 1 \wedge$
 $\mathsf{R}(v_r) = 2 \wedge \mathsf{L}(v_w) = l \wedge \mathsf{L}(v_r) = l$
 $\wedge\ \mathsf{P}(v_w) = \mathsf{true} \wedge \mathsf{P}(v_r) = \mathsf{false}$} g_initl($l$);
 {obs({}) * lock($l$) $\wedge$ I($l$)=inv($b$)}
 fork( {obs({}) * lock($l$)}
  while (true) fork(reader($b$)));
 {obs({}) * lock($l$)}
 while (true) fork(writer($b$))
 {obs({}) * lock($l$)}}

routine reader(rdwr $b$){
 {obs($O$) * lock($b.l$) $\wedge$ $b.l \preccurlyeq O \uplus \{b.v_w\}$
 $\wedge$ $b.v_r \preccurlyeq O$ $\wedge$ I($b.l$)=inv($b$)}
 acquire($b.l$);
 while($b.aw + b.ww > 0$)
   wait($b.v_r, b.l$);
 $b.ar$:=$b.ar+1$;
 g_chrg($b.v_w$);
 release($b.l$);
 // Perform reading ...
 acquire($b.l$);
 if($b.ar < 1$)
   abort;
 $b.ar$:=$b.ar-1$;
 if ($Wt(b.v_w) > 0$) g_inc($b.c_w$);
 notify($b.v_w$);
 g_disch($b.v_w$);
 release($b.l$) {obs({}) * lock($b.l$)}}

routine writer(rdwr $b$){
 {obs($O$) * lock($b.l$) $\wedge$ $b.l \preccurlyeq O \uplus \{b.v_w, b.v_r\}$
 $\wedge$ $b.v_w \preccurlyeq O \uplus \{b.v_w, b.v_r\}$ $\wedge$ I($b.l$)=inv($b$)}
 acquire($b.l$);
 g_chrg($b.v_w$); g_inc($b.c_w$);
 g_chrg($b.v_r$);
 while($b.aw + b.ar > 0$){
   g_dec($b.c_w$);
   $b.ww$:=$b.ww+1$;
   wait($b.v_w, b.l$);
   if($b.ww < 1$)
      abort();
   $b.ww$:=$b.ww-1$
 };
 $b.aw$:=$b.aw+1$;
 g_dec($b.c_w$);
 release($b.l$);
 // Perform writing ...
 acquire($b.l$);
 if($b.aw \neq 1$)
   abort;
 $b.aw$:=$b.aw-1$;
 if ($Wt(b.v_w) > 0$) g_inc($b.c_w$);
 notify($b.v_w$);
 if($b.ww = 0$)
   notifyAll($b.v_r$);
 g_disch($b.v_w$); g_disch($b.v_r$);
 release($b.l$) {obs({}) * lock($b.l$)}}
```

Fig. 11. Verification of the program in Fig. 10

writer thread, this command can be verified if $v_w \preccurlyeq \{\!\{v_w\}\!\}$, where $\mathsf{P}(v_w)$ must be true. The verification of this program is illustrated in Fig. 11. Generally, for a condition variable v for which $P(v) = \mathsf{true}$ a lock invariant can imply the invariant $\mathsf{one\_ob}(v, Wt, Ot)$ if it asserts $Wt(v) + Ct(v) < Ot(v) + S(v)$ and $Wt(v) = 0 \vee Wt(v) < Ot(v)$, where $Ct(v)$ is the total number of credits for v and $S(v)$ is an integer value such that $\mathsf{wait}(v, l)$ is executed only if $S(v) \leqslant 0$.

4.2 A Further Relaxation

The relation $\preccurlyeq$ allows one to verify some deadlock-free programs where a thread waits for a condition variable while that thread is also obliged to fulfil an obligation for that variable. However, it is still possible to have a more general, more relaxed definition for this relation. Under this definition a thread with obligations O is allowed to wait for a condition variable v if either $v \prec O$, or there exists an obligation o such that (1) $v \prec O - \{\!\{o\}\!\}$, and (2) o satisfies the property that in any execution state, if a thread is waiting for o then there exists a thread that is not waiting for any waitable object whose wait level is equal to/greater than the wait levels of v and o. This new definition still guarantees that in any state of the execution if the program has some threads suspended, waiting for some obligations, there is always a thread obliged to fulfil the obligation o_{min} that is not suspended, where o_{min} has a minimal wait level among all waitable objects for which a thread is waiting. To satisfy the condition number 2 we introduce a new definition for $\preccurlyeq$, shown in Definition 3, that uses a new function X mapping each lock to a set of wait levels. This definition will be sound only if the proof rules ensure that for any condition variable v whose wait level is in $\mathsf{X}(\mathsf{L}(\mathsf{v}))$ the number of obligations is equal to or greater than the number of the waiting threads.

This definition is still sound because of Lemma 1, that has been machine-checked in Coq[5], where G is a bag of waitable object-bag of obligations pairs such that each element t of G is associated with a thread in a state of the execution, where the first element of t is the object for which t is waiting and the second element is the bag of obligations of t. This lemma implies that if all the mentioned rules, denoted by H_1 to H_4, are respected in any state of the execution then it is impossible that all threads in that state are waiting for a waitable object. This lemma can be proved by induction on the number of elements of G and considering the element waiting for an object whose wait level is minimal (see [16] representing its proof in details).

Definition 3 (Relaxed precedence relation). *The new precedence relation indexed over functions* $\mathsf{R}, \mathsf{L}, \mathsf{P}, \mathsf{X}$ *holds for a waitable object v and a bag of obligations O, denoted by $v \preccurlyeq O$, if and only if:*

[5] The machine-checked proof can be found at https://github.com/jafarhamin/deadlock-free-monitors-soundness.

$$(v \prec O \ \vee \ v \preceq O) \wedge (\neg \mathsf{exc}(v) \vee v \perp O), \textit{ where}$$
$$v \prec O \Leftrightarrow \forall o \in O.\ \mathsf{R}(v) < \mathsf{R}(o)$$
$$v \preceq O \Leftrightarrow \mathsf{P}(v) = \mathsf{true} \wedge \mathsf{exc}(v) \wedge$$
$$\quad \exists o.\ v \prec O - \{\!\!\{o\}\!\!\} \wedge \mathsf{R}(v) \leqslant \mathsf{R}(o) + 1 \wedge \mathsf{L}(v) = \mathsf{L}(o) \wedge \mathsf{exc}(o)$$
$$\mathsf{exc}(v) = \mathsf{R}(v) \in \mathsf{X}(\mathsf{L}(v))$$
$$v \perp O \Leftrightarrow \mathsf{let}\ Ox = \lambda v'. \begin{cases} O(v') & \textit{if } \mathsf{R}(v') \in \mathsf{X}(\mathsf{L}(v)) \\ 0 & \textit{otherwise} \end{cases} \ \mathsf{in}$$
$$\quad |Ox| \leqslant 1 \wedge$$
$$\quad \forall v'.\ Ox(v') > 0 \Rightarrow \mathsf{L}(v') = \mathsf{L}(v)$$

Lemma 1 (A Valid Graph Is Not Deadlocked)

$\forall\ G{:}Bags(WaitObjs \times Bags(WaitObjs)),\ R{:}WaitObjs \to WaitLevels,$
$L{:}WaitObjs \to Locks,\ P{:}WaitObjs \to Bools,\ X{:}Locks \to Sets(WaitLevels).$
$H_1 \wedge H_2 \wedge H_3 \wedge H_4 \Rightarrow G = \{\!\!\}\!\!\},\ \textit{where}$
$\quad H_1 : \forall (o, O) \in G.\ 0 < \mathsf{Ot}(o)$
$\quad H_2 : \forall (o, O) \in G.\ P(o) = \mathsf{true} \Rightarrow \mathsf{Wt}(o) < \mathsf{Ot}(o)$
$\quad H_3 : \forall (o, O) \in G.\ R(o) \in X(L(o)) \Rightarrow \mathsf{Wt}(o) \leqslant \mathsf{Ot}(o)$
$\quad H_4 : \forall (o, O) \in G.\ o \preccurlyeq_{R,L,P,X} O$
$\textit{where } \mathsf{Wt} = \biguplus_{(o,O)\in G} \{\!\!\{o\}\!\!\} \textit{ and } \mathsf{Ot} = \biguplus_{(o,O)\in G} O$

$$\textsc{NewLock}\ \{\mathsf{true}\}\ \mathsf{newlock}\ \{\lambda l.\ \mathsf{ulock}(l, \{\!\!\}\!\!\}, \{\!\!\}\!\!\}) \wedge \mathsf{R}(l){=}z \wedge \mathsf{X}(l){=}X\}$$

$$\textsc{NewCv}\ \{\mathsf{true}\}\ \mathsf{newcond}\ \{\lambda v.\ \mathsf{R}(v){=}z \wedge \mathsf{L}(v){=}l \wedge \mathsf{M}(v){=}m \wedge \mathsf{P}(v){=}b\}$$

Fig. 12. New proof rules initializing functions X and P used in safe_obs and $\preccurlyeq$

To
extend the proof rules with the new precedence relation it suffices to include a new invariant own_ob in the definition of safe_obs, as shown in Definition 4, an invariant that must hold when a command wait or a ghost command g_disch is executed, to make sure that for any condition variable for which exc holds, the number of obligations is equal to/greater than the number of the waiting threads. Additionally, the functions X and P, as indicated in Fig. 12, are initialized when a lock and a condition variable is created, respectively. The rest of the proof rules are the same as those defined in Fig. 5 except that the old precedence relation ($\prec$) is replaced by the new one ($\preccurlyeq$).

Definition 4 (Safe Obligations). *The relation* safe_obs(v, Wt, Ot)*, indexed over functions* R, L, P, X*, holds if and only if:*

$$\mathsf{one\_ob}(v, Wt, Ot) \ \wedge \ (\mathsf{P}(v) = \mathsf{true} \Rightarrow \mathsf{spare\_ob}(v, Wt, Ot)) \wedge$$
$$(\mathsf{exc}(v) = \mathsf{true} \Rightarrow \mathsf{own\_ob}(v, Wt, Ot)), \textit{ where}$$
$$\mathsf{one\_ob}(v, Wt, Ot) \Leftrightarrow (Wt(v) > 0 \Rightarrow Ot(v) > 0)$$
$$\mathsf{spare\_ob}(v, Wt, Ot) \Leftrightarrow (Wt(v) > 0 \Rightarrow Wt(v) < Ot(v))$$
$$\mathsf{own\_ob}(v, Wt, Ot) \Leftrightarrow (Wt(v) \leqslant Ot(v))$$

Bounded Channels. One application of the new definition is a bounded channel program, shown in Fig. 13, where a sender thread waits for a receiver thread if the channel is full, synchronized by v_f, and a receiver thread waits for a sender thread if the channel is empty, synchronized by v_e. More precisely, the sender thread with an obligation for v_e might execute the command wait(v_f, l), and the receiver thread with an obligation for v_f might execute a command wait(v_e, l).

```
routine main(){
 q := newqueue;
 l := newlock;
 v_f := newcvar;
 v_e := newcvar;
 ch:=channel(q, l, v_f, v_e);
 fork (receive(ch));
 send(ch, 12)}
```

```
routine send(channel ch, int d)
{
 acquire(ch.l);
 while(sizeof(ch.q) = max)
   wait(ch.v_f, ch.l);
 enqueue(ch.q, d);
 notify(ch.v_e);
 release(ch.l)}
```

```
routine receive(channel ch)
{
 acquire(ch.l);
 while(sizeof(ch.q) = 0)
   wait(ch.v_e, ch.l);
 dequeue(ch.q);
 notify(ch.v_f);
 release(ch.l)}
```

```
inv(channel ch) ::= $\lambda Wt.\ \lambda Ot.\ \exists Cte, Ctf.$ ctr$(ch.c_e, Cte) *$ ctr$(ch.c_f, Ctf) *$
 $\exists s.$ queue$(ch.q, s) \wedge$ P$(v_e)$=false $\wedge$ M$(v_e)$=tic$(ch.c_e) \wedge$ M$(v_f)$=tic$(ch.c_f)$ land
 L$(ch.v_e)$=L$(ch.v_f)$=$ch.l \wedge$
  $Wt(ch.v_e) + Cte \leqslant Ot(ch.v_e) + s \wedge Wt(ch.v_e) \leqslant Ot(ch.v_e) \wedge$
  $Wt(ch.v_f) + Ctf + s < Ot(ch.v_f) +$ max $\wedge$ $(Wt(v_f) = 0 \vee Wt(ch.v_f) < Ot(ch.v_f))$
```

```
routine main(){
 q := newqueue;
 l := newlock;
 v_f := newcvar;
 v_e := newcvar;
 ch:=channel(q, l, v_f, v_e);
 ch.c_e:=g_newctr;
 ch.c_f:=g_newctr;
 g_inc(ch.c_e);
 g_inc(ch.c_f);
 g_chrg(v_e);  g_chrg(v_f);
 g_initl(l);
 {obs({v_e, v_f}) * lock(l) *
 tic(ch.c_e) * tic(ch.c_f) *
 L(v_f)=l ∧ L(v_e)=l ∧
 M(v_e)=tic(ch.c_e) ∧
 M(v_f)=tic(ch.c_f) ∧
 P(v_f)=true ∧
 P(v_e)=false ∧
 R(l)=0 ∧
 R(v_e)=1 ∧ R(v_f)=2 ∧
 X(l)={1, 2} ∧ I(l)=inv}
 fork (receive(ch));
 send(ch, 12) {obs({})}}
```

```
routine send(channel ch, int d)
{{obs(O⊎{ch.v_e}) * tic(ch.c_f) *
 lock(ch.l) ∧ ch.l≺O⊎{ch.v_e} ∧
 ch.v_f≺O⊎{ch.v_e}∧I(ch.l)=inv}
 acquire(ch.l);
 while(sizeof(ch.q) = max){
   g_dec(ch.c_f);
   wait(ch.v_f, ch.l)};
 enqueue(ch.q, d);
 if ( Wt(b.v_e) > 0)
   g_inc(b.c_e);
 notify(ch.v_e);
 g_disch(ch.v_e);
 g_dec(ch.c_f);
 release(ch.l)
 {obs(O) * lock(ch.l)}}
```

```
routine receive(channel ch){
{obs(O⊎{ch.v_f}) * tic(ch.c_e) *
 lock(ch.l) ∧ ch.l≺O⊎{ch.v_f} ∧
 ch.v_e≺O⊎{ch.v_f}∧I(ch.l)=inv}
 acquire(ch.l);
 while(sizeof(ch.q) = 0){
   g_dec(ch.c_e);
   wait(ch.v_e, ch.l)};
 dequeue(ch.q);
 if ( Wt(b.v_f) > 0)
   g_inc(b.c_f);
 notify(ch.v_f);
 g_disch(ch.v_f);
 g_dec(ch.c_e);
 release(ch.l)
 {obs(O) * lock(ch.l)}}
```

Fig. 13. Verification of a bounded channel synchronized using a monitor consisting of condition variables v_f, preventing sending on a full channel, and v_e, preventing taking messages from an empty channel

Since v_e and v_f are not equal, it is impossible to verify this program by the old definition of $\preccurlyeq$ because the waiting levels of v_e and v_f cannot be lower than each other. Thanks to the new definition of $\preccurlyeq$, this program can be verified, as shown in Fig. 13, by initializing $\mathsf{P}(v_f)$ with true and $\mathsf{X}(l)$ with $\{1, 2\}$, where two consecutive numbers 1 and 2 are the wait levels of v_e and v_f, respectively.

5 Soundness Proof

In this section we provide a soundness proof for the present approach[6], i.e. if a program is verified by the proposed proof rules, where the verification starts from an empty bag of obligations and also ends with such bag, this program is deadlock-free. To this end, we first define the syntax of programs and a small-step semantics for programs ($\rightsquigarrow$) relating two *configurations* (see [16] for formal definitions). A configuration is a thread table-heap pair (t, h), where heaps and thread tables are some partial functions from locations and thread identifiers to integers and command-*context* pairs $(c; \xi)$, respectively, where a context, denoted by ξ, is either done or $\mathsf{let}\ x{:=}[]\ \mathsf{in}\ c; \xi$. Then we define *validity of configurations*, shown in Definition 5, and prove that (1) if a program c is verified by the proposed proof rules, where it starts from the precondition $\mathsf{obs}(\{\!\!\}\!\!\})$ and satisfies the post condition $\lambda\_.\mathsf{obs}(\{\!\!\}\!\!\})$, then the initial configuration, where the heap is empty, denoted by $\mathbf{0} = \lambda\_.\varnothing$, and there is only one thread with command c and context done, is a valid configuration (Theorem 4), (2) a valid configuration is not deadlocked (Theorem 5), and (3) starting from a valid configuration, all the subsequent configurations of the execution are also valid (Theorem 6).

In a valid configuration (t, h), h contains all the heap ownerships that are in possession of all threads in t and also those that are in possession of the locks that are not held, specified by a list A. Additionally, each thread must have all the required permissions to be successfully verified with no remaining obligation, enforced by wpcx. $\mathsf{wpcx}(c, \xi)$ in this definition is a function returning the weakest precondition of the command c with the context ξ w.r.t. the postcondition $\lambda\_.\mathsf{obs}(\{\!\!\}\!\!\})$ (see [16] for formal definitions). This function is defined with the help of a function $\mathsf{wp}(c, a)$ returning the weakest precondition the command c w.r.t. the postcondition a.

Definition 5 (Validity of Configurations). *A configuration is valid, denoted by* $\mathsf{valid}(t, h)$*, if there exist a list of* augmented threads T*, consisting of an identifier (id), a program (c), a context (ξ), a permission heap (p), a ghost resource heap (C) and a bag of obligations (O) associated with each thread; a list of assertions A, and some functions R, I, L, M, P, X such that:*

– $\forall id, c, \xi.\ t(id) = (c; \xi) \Leftrightarrow \exists p, O, C.\ (id, c, \xi, p, O, C) \in T$
– $h = \mathsf{pheap2heap}(\mathop{*}\limits_{a \in A} a \ * \mathop{*}\limits_{(id,c,\xi,p,O,C) \in T} p)$

[6] The machine-checked version of some lemmas and theorems in this proof, such as Theorems 4 and 5, can be found at https://github.com/jafarhamin/deadlock-free-monitors-soundness.

– $\forall (id, c, \xi, p, O, C) \in T$.
  - $p, O, C \models \mathsf{wpcx}_{R,I,L,M,P,X}(c, \xi)$
  - $\forall l, Wt, Ot.\ p(l) = \mathsf{Ulock/Locked}(Wt, Ot) \Rightarrow Wt = \mathsf{Wt}_l \wedge Ot = \mathsf{Ot}_l$
  - $\forall l.\ p(l) = \mathsf{Lock} \wedge h(l) = 1 \Rightarrow \mathsf{I}(l)(\mathsf{Wt}_l, \mathsf{Ot}_l) \in A$
  - $\forall l.\ p(l) = \mathsf{Lock} \vee p(l) = \mathsf{Locked}(\mathsf{Wt}_l, \mathsf{Ot}_l) \Rightarrow \neg P(l) \wedge \neg \mathsf{exc}(l) \wedge (h(l) = 0 \Rightarrow l \in \mathsf{Ot})$
  - $\forall o.\ \mathsf{waiting\_for}(c, h) = o \Rightarrow \mathsf{safe\_obs}_{R,L,P,X}(o, \mathsf{Wt}, \mathsf{Ot})$

where

- $\mathsf{Ot} = \biguplus_{(id,c,\xi,p,O,C) \in T} O, \mathsf{Wt} = \biguplus_{(id,c,\xi,p,O,C) \in T \wedge \mathsf{waiting\_for}(c,h)=o} \{\!\!\{o\}\!\!\}$
- O_l *is a bag that given an object* o *returns* $O(o)$ *if* $L(o) = l$ *and* 0 *if* $L(o) \neq l$
- $\mathsf{waiting\_for}(c, h)$ *returns the object for which* c *is waiting, if any*
- $\mathsf{pheap2heap}(p)$ *returns the heap corresponding with permission heap* p

We finally prove that for each proof rule $\{a\}\ c\ \{a'\}$ we have $a \Rightarrow \mathsf{wp}(c, a')$. To this end, we first define *correctness of commands*, shown in Definition 6, and then for each proof rule $\{a\}\ c\ \{a'\}$ we prove $\mathsf{correct}(a, c, a')$. In addition to the proof rules presented in this paper, other useful rules such as the rules *consequence*, *frame* and *sequential*, shown in Theorems 1, 2, and 3 can also be proved with the help of some auxiliary lemmas in [16]. Note that the indexes R, I, L, M, P, X are omitted when they are unimportant.

Definition 6 (Correctness of Commands)

$$\mathsf{correct}_{R,I,L,M,P,X}(a, c, a') \Leftrightarrow (a \Rightarrow \mathsf{wp}_{R,I,L,M,P,X}(c, a'))$$

Theorem 1 (Rule Consequence)

$$\mathsf{correct}(a_1, c, a_2) \wedge (a'_1 \Rightarrow a_1) \wedge (\forall z.\ a_2(z) \Rightarrow a'_2(z)) \Rightarrow \mathsf{correct}(a'_1, c, a'_2)$$

Theorem 2 (Rule Frame)

$$\mathsf{correct}(a, c, a') \Rightarrow \mathsf{correct}(a * f, c, \lambda z.\ a'(z) * f)$$

Theorem 3 (Rule Sequential Composition)

$$\mathsf{correct}(a, c_1, a') \wedge (\forall z.\ \mathsf{correct}(a'(z), c_2[z/x], a'')) \Rightarrow$$
$$\mathsf{correct}(a, \mathsf{let}\ x{:=}c_1\ \mathsf{in}\ c_2, a'')$$

Theorem 4 (The Initial Configuration is Valid)

$$\mathsf{correct}_{R,I,L,M,P,X}(\mathsf{obs}(\{\!\!\{\}\!\!\}), c, \lambda\_.\mathsf{obs}(\{\!\!\{\}\!\!\})) \Rightarrow \mathsf{valid}(\mathbf{0}[id{:=}c; \mathsf{done}], \mathbf{0})$$

Proof. The goal is achieved because there are an augmented thread list $T = [(id, c, \mathsf{done}, \mathbf{0}, \{\!\!\{\}\!\!\}, \mathbf{0})]$, a list of assertions $A = []$, and functions R, I, L, M, P, X by which all the conditions in the definition of validity of configurations are satisfied.

Theorem 5 (A Valid Configuration is Not Deadlocked)

$$(\exists id, c, \xi, o.\ t(id) = (c;\xi) \wedge \mathsf{waiting\_for}(c,h) = o) \wedge \mathsf{valid}(t,h)$$
$$\Rightarrow \exists id', c', \xi',\ t(id') = (c';\xi') \wedge \mathsf{waiting\_for}(c',h) = \varnothing$$

Proof. We assume that all threads in t are waiting for an object. Since (t,h) is a valid configuration there exists a valid augmented thread table T with a corresponding valid graph $G = \mathsf{g}(T)$, where g maps any element such as (id, c, ξ, p, O, C) to a new one such as $(\mathsf{waiting\_for}(c), O)$. By Lemma 1, we have $G = \{\!\!\}$, implying $T = \{\!\!\}$, implying $t = \mathbf{0}$ which contradicts the assumption of the theorem.

Theorem 6 (Steps Preserve Validity of Configurations).[7]

$$\mathsf{valid}(\kappa) \wedge \kappa \rightsquigarrow \kappa' \Rightarrow \mathsf{valid}(\kappa')$$

Proof. By case analysis of the small step relation $\rightsquigarrow$ (see [16] explaining the proof of some non-trivial cases).

6 Related Work

Several approaches to verify termination [1,21], total correctness [3], and lock freedom [2] of concurrent programs have been proposed. These approaches are only applicable to non-blocking algorithms, where the suspension of one thread cannot lead to the suspension of other threads. Consequently, they cannot be used to verify deadlock-freedom of programs using condition variables, where the suspension of a notifying thread might lead a waiting thread to be infinitely blocked. In [22] a compositional approach to verify termination of multi-threaded programs is introduced, where *rely-guarantee reasoning* is used to reason about each thread individually while there are some assertions about other threads. In this approach a program is considered to be terminating if it does not have any infinite computations. As a consequence, it is not applicable to programs using condition variables because a waiting thread that is never notified cannot be considered as a terminating thread.

There are also some other approaches addressing some common synchronization bugs of programs in the presence of condition variables. In [8], for example, an approach to identify some potential problems of concurrent programs consisting waits and notifies commands is presented. However, it does not take the order of execution of theses commands into account. In other words, it might accept an undesired execution trace where the waiting thread is scheduled before the notifying thread, that might lead the waiting thread to be infinitely suspended. [9] uses Petri nets to identify some common problems in multithreaded programs such as data races, lost signals, and deadlocks. However the model introduced for condition variables in this approach only covers the communication of two threads and it is not clear how it deals with programs having

[7] The proof of this theorem has not been machine-checked with Coq yet.

more than two threads communicating through condition variables. Recently, [10] has introduced an approach ensuring that every thread synchronizing under a set of condition variables eventually exits the synchronization block if that thread eventually reaches that block. This approach succeeds in verifying one of the applications of condition variables, namely the buffer. However, since this approach is not modular and relies on a Petri net analysis tool to solve the termination problem, it suffers from a long verification time when the size of the state space is increased, such that the verification of a buffer application having 20 producer and 18 consumer threads, for example, takes more than two minutes.

Kobayashi [6,20] proposed a type system for deadlock-free processes, ensuring that a well-typed process that is annotated with a finite *capability level* is deadlock free. He extended channel types with the notion of *usages*, describing how often and in which order a channel is used for input and output. For example, usage of x in the process $x?y|x!1|x!2$, where $?, !, |$ represent an input action, an output action, and parallel composition receptively, is expressed by $?|!|!$, which means that x is used once for input and twice for output possibly in parallel. Additionally, to avoid circular dependency each action α is associated with the levels of obligation o and capabilities c, denoted by α^o_c, such that (1) an obligation of level n must be fulfilled by using only capabilities of level less than n, and (2) for an action of capability level n, there must exist a co-action of obligation level less than or equal to n. Leino *et al.* [4] also proposed an approach to verify deadlock-freedom of channels and locks. In this approach each thread trying to receive a message from a channel must spend one credit for that channel, where a credit for a channel is obtained if a thread is obliged to fulfil an obligation for that channel. A thread can fulfil an obligation for a channel if either it sends a message on that channel or delegate that obligation to other thread. The same idea is also used to verify deadlock-freedom of semaphores [7], where acquiring (i.e. decreasing) a semaphore consumes one credit and releasing (i.e. increasing) that semaphore produces one credit for that semaphore. However, as it is acknowledged in [4], it is impossible to treat channels (and also semaphores) like condition variables; a wait cannot be treated like a receive and a notify cannot be treated like a send because a notification for a condition variable will be lost if no thread is waiting for that variable. We borrow many ideas, including the notion of obligations/credits(capabilities) and levels, from these works and also the one introduced in [11], where a corresponding separation logic based approach is presented to verify total correctness of programs in the presence of channels.

7 Conclusion

It this article we introduced a modular approach to verify deadlock-freedom of monitors. We also introduced a relax, more general precedence relation to avoid cycles in the wait-for graph of programs, allowing a verification approach to verify a wider range of deadlock-free programs in the presence of monitors, channels and other synchronization mechanisms.

Acknowledgements. This work was funded through Flemish Research Fund grant G.0058.13 and KU Leuven Research Fund grant OT/13/065. We thank three anonymous reviewers and Prof. Aleksandar Nanevski for their careful reading of our manuscript and their many insightful comments and suggestions.

References

1. Liang, H., Feng, X., Shao, Z.: Compositional verification of termination-preserving refinement of concurrent programs. In: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), ACM (2014), Article No. 65
2. Hoffmann, J., Marmar, M., Shao, Z.: Quantitative reasoning for proving lock-freedom. In: 28th Annual IEEE/ACM Symposium on Logic in Computer Science (LICS). IEEE, pp. 124–133 (2013)
3. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P., Sutherland, J.: Modular termination verification for non-blocking concurrency. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 176–201. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_8
4. Leino, K.R.M., Müller, P., Smans, J.: Deadlock-free channels and locks. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 407–426. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_22
5. Boström, P., Müller, P.: Modular Verification of Finite Blocking in Non-terminating Programs, vol. 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Germany (2015)
6. Kobayashi, N.: A new type system for deadlock-free processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006). https://doi.org/10.1007/11817949_16
7. Jacobs, B.: Provably live exception handling. In: Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs. ACM (2015) Article No. 7
8. Wang, C., Hoang, K.: Precisely deciding control state reachability in concurrent traces with limited observability. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 376–394. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_21
9. Kavi, K.M., Moshtaghi, A., Chen, D.J.: Modeling multithreaded applications using petri nets. Int. J. Parallel Prog. **30**(5), 353–371 (2002)
10. de Carvalho Gomes, P., Gurov, D., Huisman, M.: Specification and verification of synchronization with condition variables. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2016. CCIS, vol. 694, pp. 3–19. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-53946-1_1
11. Jacobs, B., Bosnacki, D., Kuiper, R.: Modular termination verification. In: LIPIcs-Leibniz International Proceedings in Informatics, vol. 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Germany (2015)
12. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. NASA Form. Methods **6617**, 41–55 (2011)
13. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17164-2_21

14. Jacobs, B., (ed.): VeriFast 18.02. Zenodo (2018). https://doi.org/10.5281/zenodo.1182724
15. Dijkstra, E.W.: Cooperating sequential processes. The origin of concurrent programming, pp. 65–138. Springer, New York (1968)
16. Hamin, J., Jacobs, B.: Deadlock-free monitors: extended version. Technical report CW712, Department of Computer Science, KU Leuven, Belgium (2018)
17. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. ACM SIGPLAN Not. **46**(1), 271–282 (2011)
18. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings on 17th Annual IEEE Symposium on Logic in Computer Science, IEEE, pp. 55–74 (2002)
19. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. ACM SIGPLAN Not. **50**(1), 637–650 (2015)
20. Kobayashi, N.: Type systems for concurrent programs. In: Aichernig, B.K., Maibaum, T. (eds.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 439–453. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40007-3_26
21. Hamin, J., Jacobs, B.: Modular verification of termination and execution time bounds using separation logic. In: 17th International Conference on Information Reuse and Integration (IRI), IEEE, pp. 110–117 (2016)
22. Popeea, C., Rybalchenko, A.: Compositional termination proofs for multi-threaded programs. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 237–251. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_17
23. Vafeiadis, V.: Concurrent separation logic and operational semantics. Electron. Notes Theor. Comput. Sci. **276**, 335–351 (2011)

Fragment Abstraction for Concurrent Shape Analysis

Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh(✉)

Uppsala University, Uppsala, Sweden
cong-quy.trinh@it.uu.se

Abstract. A major challenge in automated verification is to develop techniques that are able to reason about fine-grained concurrent algorithms that consist of an unbounded number of concurrent threads, which operate on an unbounded domain of data values, and use unbounded dynamically allocated memory. Existing automated techniques consider the case where shared data is organized into singly-linked lists. We present a novel shape analysis for automated verification of fine-grained concurrent algorithms that can handle heap structures which are more complex than just singly-linked lists, in particular skip lists and arrays of singly linked lists, while at the same time handling an unbounded number of concurrent threads, an unbounded domain of data values (including timestamps), and an unbounded shared heap. Our technique is based on a novel shape abstraction, which represents a set of heaps by a set of *fragments*. A fragment is an abstraction of a pair of heap cells that are connected by a pointer field. We have implemented our approach and applied it to automatically verify correctness, in the sense of linearizability, of most linearizable concurrent implementations of sets, stacks, and queues, which employ singly-linked lists, skip lists, or arrays of singly-linked lists with timestamps, which are known to us in the literature.

1 Introduction

Concurrent algorithms with an unbounded number of threads that concurrently access a dynamically allocated shared state are of central importance in a large number of software systems. They provide efficient concurrent realizations of common interface abstractions, and are widely used in libraries, such as the Intel Threading Building Blocks or the `java.util.concurrent` package. They are notoriously difficult to get correct and verify, since they often employ fine-grained synchronization and avoid locking when possible. A number of bugs in published algorithms have been reported [13,30]. Consequently, significant research efforts have been directed towards developing techniques to verify correctness of such algorithms. One widely-used correctness criterion is that of *linearizability*, meaning that each method invocation can be considered to occur atomically at some point between its call and return. Many of the developed verification techniques require significant *manual* effort for constructing correctness

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 442–471, 2018.
https://doi.org/10.1007/978-3-319-89884-1_16

proofs (e.g., [25,41]), in some cases with the support of an interactive theorem prover (e.g., [11,35,40]). Development of automated verification techniques remains a difficult challenge.

A major challenge for the development of automated verification techniques is that such techniques must be able to reason about fine-grained concurrent algorithms that are infinite-state in many dimensions: they consist of an unbounded number of concurrent threads, which operate on an unbounded domain of data values, and use unbounded dynamically allocated memory. Perhaps the hardest of these challenges is that of handling dynamically allocated memory. Consequently, existing techniques that can automatically prove correctness of such fine-grained concurrent algorithms restrict attention to the case where heap structures represent shared data by singly-linked lists [1,3,18,36,42]. Furthermore, many of these techniques impose additional restrictions on the considered verification problem, such as bounding the number of accessing threads [4,43,45]. However, in many concurrent data structure implementations the heap represents more sophisticated structures, such as skiplists [16,22,38] and arrays of singly-linked lists [12]. There are no techniques that have been applied to automatically verify concurrent algorithms that operate on such data structures.

Contributions. In this paper, we present a technique for automatic verification of concurrent data structure implementations that operate on dynamically allocated heap structures which are more complex than just singly-linked lists. Our framework is the first that can automatically verify concurrent data structure implementations that employ singly linked lists, skiplists [16,22,38], as well as arrays of singly linked lists [12], at the same time as handling an unbounded number of concurrent threads, an unbounded domain of data values (including timestamps), and an unbounded shared heap.

Our technique is based on a novel shape abstraction, called *fragment abstraction*, which in a simple and uniform way is able to represent several different classes of unbounded heap structures. Its main idea is to represent a set of heap states by a set of *fragments*. A fragment represents two heap cells that are connected by a pointer field. For each of its cells, the fragment represents the contents of its non-pointer fields, together with information about how the cell can be reached from the program's global pointer variables. The latter information consists of both: (i) *local* information, saying which pointer variables point directly to them, and (ii) *global* information, saying how the cell can reach to and be reached from (by following chains of pointers) heap cells that are globally significant, typically since some global variable points to them. A set of fragments represents the set of heap states in which any two pointer-connected nodes is represented by some fragment in the set. Thus, a set of fragments describes the set of heaps that can be formed by "piecing together" fragments in the set. The combination of local and global information in fragments supports reasoning about the sequence of cells that can be accessed by threads that traverse the heap by following pointer fields in cells and pointer variables: the local information captures properties of the cell fields that can be accessed as a thread dereferences a pointer variable or a pointer field; the global information also captures whether certain significant accesses will at all be possible by following a sequence of pointer fields. This support for reasoning about patterns of cell accesses enables automated verification of reachability and other functional properties.

Fragment abstraction can (and should) be combined, in a natural way, with data abstractions for handling unbounded data domains and with thread abstractions for handling an unbounded number of threads. For the latter we adapt the successful thread-modular approach [5], which represents the local state of a single, but arbitrary thread, together with the part of the global state and heap that is accessible to that thread. Our combination of fragment abstraction, thread abstraction, and data abstraction results in a finite abstract domain, thereby guaranteeing termination of our analysis.

We have implemented our approach and applied it to automatically verify correctness, in the sense of linearizability, of a large number of concurrent data structure algorithms, described in a C-like language. More specifically, we have automatically verified linearizability of most linearizable concurrent implementations of sets, stacks, and queues, and priority queues, which employ singly-linked lists, skiplists, or arrays of timestamped singly-linked lists, which are known to us in the literature on concurrent data structures. For this verification, we specify linearizability using the simple and powerful technique of *observers* [1,7,9], which reduces the criterion of linearizability to a simple reachability property. To verify implementations of stacks and queues, the application of observers can be done completely automatically without any manual steps, whereas for implementations of sets, the verification relies on light-weight user annotation of how linearization points are placed in each method [3].

The fact that our fragment abstraction has been able to automatically verify all supplied concurrent algorithms, also those that employ skiplists or arrays of SLLs, indicates that the fragment abstraction is a simple mechanism for capturing both the local and global information about heap cells that is necessary for verifying correctness, in particular for concurrent algorithms where an unbounded number of threads interact via a shared heap.

Outline. In the next section, we illustrate our fragment abstraction on the verification of a skiplist-based concurrent set implementation. In Sect. 3 we introduce our model for programs, and of observers for specifying linearizability. In Sect. 4 we describe in more detail our fragment abstraction for skiplists; note that singly-linked lists can be handled as a simple special case of skiplists. In Sect. 5 we describe how fragment abstraction applies to arrays of singly-linked lists with timestamp fields. Our implementation and experiments are reported in Sect. 6, followed by conclusions in Sect. 7.

Related Work. A large number of techniques have been developed for representing heap structures in automated analysis, including, e.g., separation logic and various related graph formalisms [10,15,47], other logics [33], automata [23], or graph grammars [19]. Most works apply these to sequential programs.

Approaches for automated verification of concurrent algorithms are limited to the case of singly-linked lists [1,3,18,36,42]. Furthermore, many of these techniques impose additional restrictions on the considered verification problem, such as bounding the number of accessing threads [4,43,45].

In [1], concurrent programs operating on SLLs are analyzed using an adaptation of a transitive closure logic [6], combined with tracking of simple sortedness properties between data elements; the approach does not allow to represent patterns observed by threads when following sequences of pointers inside the heap, and so has not been applied to concurrent set implementations. In our recent work [3], we extended this approach to handle SLL implementations of concurrent sets by adapting a well-known abstraction of singly-linked lists [28] for concurrent programs. The resulting technique is specifically tailored for singly-links. Our fragment abstraction is significantly simpler conceptually, and can therefore be adapted also for other classes of heap structures. The approach of [3] is the only one with a shape representation strong enough to verify concurrent set implementations based on sorted and non-sorted singly-linked lists having non-optimistic contains (or lookup) operations we consider, such as the lock-free sets of *HM* [22], *Harris* [17], or *Michael* [29], or unordered set of [48]. As shown in Sect. 6, our fragment abstraction can handle them as well as also algorithms employing skiplists and arrays of singly-linked lists.

There is no previous work on automated verification of skiplist-based concurrent algorithms. Verification of *sequential* algorithms have been addressed under restrictions, such as limiting the number of levels to two or three [2,23]. The work [34] generates verification conditions for statements in sequential skiplist implementations. All these works assume that skiplists have the well-formedness property that any higher-level lists is a sublist of any lower-level list, which is true for sequential skiplist algorithms, but false for several concurrent ones, such as [22,26].

Concurrent algorithms based on arrays of SLLs, and including timestamps, e.g., for verifying the algorithms in [12] have shown to be rather challenging. Only recently has the TS stack been verified by non-automated techniques [8] using a non-trivial extension of forward simulation, and the TS queue been verified manually by a new technique based on partial orders [24,37]. We have verified both these algorithms automatically using fragment abstraction.

Our fragment abstraction is related in spirit to other formalisms that abstract dynamic graph structures by defining some form of equivalence on its nodes (e.g., [23,33,46]). These have been applied to verify functional correctness fine-grained concurrent algorithms for a limited number of SLL-based algorithms. Fragment abstraction's representation of both local and global information allows to extend the applicability of this class of techniques.

2 Overview

In this section, we illustrate our technique on the verification of correctness, in the sense of linearizability, of a concurrent set data structure based on skiplists, namely the Lock-Free Concurrent Skiplist from [22, Sect. 14.4]. Skiplists provide expected logarithmic time search while avoiding some of the complications of tree structures. Informally, a skiplist consists of a collection of sorted linked lists, each of which is located at a *level*, ranging from 1 up to a maximum value. Each skiplist node has a key value and participates in the lists at levels 1 up to its *height*. The skiplist has sentinel head and tail nodes with maximum heights and key values $-\infty$ and $+\infty$, respectively. The lowest-level list (at level 1) constitutes an ordered list of all nodes in the skiplist. Higher-level lists are increasingly sparse sublists of the lowest-level list, and serve as shortcuts into lower-level lists. Figure 1 shows an example of a skiplist of height 3. It has head and tail nodes of height 3, two nodes of height 2, and one node of height 1.

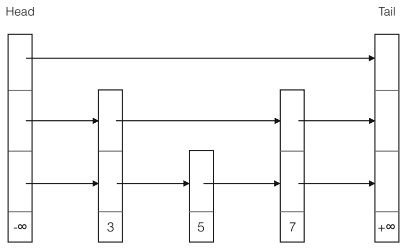


Fig. 1. An example of skiplist

The algorithm has three main methods, namely `add`, `contains` and `remove`. The method `add`(x) adds `x` to the set and returns true iff `x` was not already in the set; `remove`(x) removes `x` from the set and returns true iff `x` was in the set; and `contains`(x) returns true iff `x` is in the set. All methods rely on a method `find` to search for a given key. In this section, we shortly describe the `find` and `add` methods. Figure 2 shows code for these two methods.

In the algorithm, each heap node has a `key` field, a `height`, an array of `next` pointers indexed from 1 up to its `height`, and an array of `marked` fields which are true if the node has been logically removed at the corresponding level. Removal of a node (at a certain level `k`) occurs in two steps: first the node is logically removed by setting its `marked` flag at level `k` to `true`, thereafter the node is physically removed by unlinking it from the level-`k` list. The algorithm must be able to update the `next`[k] pointer and `marked`[k] field together as one atomic operation; this is standardly implemented by encoding them in a single word. The head and tail nodes of the skiplist are pointed to by global pointer variables `H` and `T`, respectively. The `find` method traverses the list at decreasing levels using two local variables `pred` and `curr`, starting at the head and at the maximum level (lines 5–6). At each level `k` it sets `curr` to `pred.next`[k] (line 7). During the traversal, the pointer variable `succ` and boolean variable `marked` are atomically assigned the values of `curr.next`[k] and `curr.marked`[k], respectively (line 9, 14). After that, the method repeatedly removes marked nodes at the current level (lines 10 to 14). This is done by using a `CompareAndSwap` (`CAS`) command (line 11), which tests whether `pred.next`[k] and `pred.marked`[k] are equal to `curr` and `false` respectively. If this test succeeds, it replaces them with `succ` and `false` and returns `true`; otherwise, the `CAS` returns `false`. During the traversal at level `k`, `pred` and `curr` are advanced until `pred` points to a node with the largest key at level `k` which is smaller than `x` (lines 15–18). Thereafter, the resulting values of `pred` and `curr` are recorded into `preds`[k] and `succs`[k] (lines 19, 20), whereafter traversal continues one level below until it reaches the bottom level. Finally, the method returns `true` if the `key` value of `curr` is equal to `x`; otherwise, it returns `false` meaning that a node with key `x` is not found.

```
struct Node { int key; int height; Node next[]; boolean marked[];}
```

```
boolean find(int x,Node preds[],Node succs[])
1  boolean marked = false;
2  boolean s;
3  retry:
4  while (true)
5    pred = H;
6    for (int k = MAXLEVEL; k >= 1; k--)
7      curr = pred.next[k];
8      while (true)
9        <succ, marked> =
                 <curr.next[k], curr.marked[k]>;
10        while (marked)
11          s=CAS(<pred.next[k],pred.marked[k]>
                   ,<curr,false>,<succ,false>);
12          if (!s) goto retry;
13          curr = pred.next[k]; ●
14          <succ, marked =
              <curr.next[k], curr.marked[k]>;
15        if (curr.key < x)
16          pred = curr;
17          curr = succ;
18        else break;
19      preds[k] = pred;
20      succs[k] = curr;
21    return (curr.key == x);
```

```
boolean add (int x):
1  int h = randomLevel;
2  Node* preds[1..h]; succs[1..h]
3  while (true);
4    if find(x,preds,succs)
5      return false;
6    else
7      Node* n = new Node(x, h);
8      for (int k = 1;k <= h; k++)
9        <n.next[k],n.marked[k]> =
                          <succ[k],false>;
10      Node* pred = preds[1];
11      Node* succ = succs[1];
12      <n.next[1],n.marked[1]>=<succ,false>
13      if !CAS(<pred.next[1],pred.marked[1]>
                ,<succ,false>,<n,false>);
14        goto 3;
15      else ●
16        for (int k = 2; k <= h; k++)
17          while (true);
18            pred = preds[k];
19            succ = succs[k];
20            if CAS(<pred.next[k],pred.marked
                 [k]>,<succ,false>,<n,false>)
21              break;
22            find(x,preds,succs);
23  return true;
```

Fig. 2. Code for the `find` and `add` methods of the skiplist algorithm. (Color figure online)

The `add` method uses `find` to check whether a node with key `x` is already in the list. If so it returns `false`; otherwise, a new node is created with randomly chosen height `h` (line 7), and with `next` pointers at levels from 1 to `h` initialised to corresponding elements of `succ` (line 8 to 9). Thereafter, the new node is added into the list by linking it into the bottom-level list between the `preds`[1] and `succs`[1] pointers returned by `find`. This is achieved by using a `CAS` to make `preds`[1].`next`[1] point to the new node (line 13). If the `CAS` fails, the `add` method will restart from the beginning (line 3) by calling `find` again, etc. Otherwise, `add` proceeds with linking the new node into the list at increasingly higher levels (lines 16 to 22). For each higher level `k`, it makes `preds`[`k`].`next`[`k`] point to the new node if it is still valid (line 20); otherwise `find` is called again to recompute `preds`[`k`] and `succs`[`k`] on the remaining unlinked levels (line 22). Once all levels are linked, the method returns `true`.

To prepare for verification, we add a specification which expresses that the skiplist algorithm of Fig. 2 is a linearizable implementation of a set data structure, using the technique of *observers* [1,3,7,9]. For our skiplist algorithm, the user first instruments statements in each method that correspond to linearization points (LPs), so that their execution announces the corresponding atomic set operation. In Fig. 2, the LP of a successful `add` operation is at line 15 of the `add` method (denoted by a blue dot) when the `CAS` succeeds, whereas the LP of an unsuccessful `add` operation is at line 13 of the `find` method (denoted by a red dot). We must now verify that in any concurrent execution of a collection of method calls, the sequence of announced operations satisfies the semantics of the set data structure. This check is performed by an *observer*, which monitors the sequence of announced operations. The observer for the set data structure utilizes a register, which is initialized with a single, arbitrary `key` value. It checks that operations on this particular value follow set semantics, i.e., that successful `add` and `remove` operations on an element alternate and that `contains` are consistent with them. We form the cross-product of the program and the observer, synchronizing on operation announcements. This reduces the problem of checking linearizability to the problem of checking that in this cross-product, regardless of the initial observer register value, the observer cannot reach a state where the semantics of the set data structure has been violated.

To verify that the observer cannot reach a state where a violation is reported, we compute a symbolic representation of an invariant that is satisfied by all reachable configurations of the cross-product of a program and an observer. This symbolic representation combines thread abstraction, data abstraction and our novel *fragment abstraction* to represent the heap state. Our *thread abstraction* adapts the thread-modular approach by representing only the view of single, but arbitrary, thread `th`. Such a view consists of the local state of thread `th`, including the value of the program counter, the state of the observer, and the part of the heap that is accessible to thread `th` via pointer variables (local to `th` or global). Our *data abstraction* represents variables and cell fields that range over small finite domains by their concrete values, whereas variables and fields that range over the same domain as `key` fields are abstracted to constraints over their relative ordering (wrp. to $<$).

In our *fragment abstraction*, we represent the part of the heap that is accessible to thread `th` by a set of *fragments*. A fragment represents a pair of heap cells (accessible to `th`) that are connected by a pointer field, under the applied data abstraction. A fragment is a triple of form $\langle \mathtt{i}, \mathtt{o}, \phi \rangle$, where `i` and `o` are *tags* that represent the two cells, and ϕ is a subset of $\{<, =, >\}$ which constrains the order between the `key` fields of the cells. Each tag is a tuple $\mathtt{tag} = \langle \mathtt{dabs}, \mathtt{pvars}, \mathtt{reachfrom}, \mathtt{reachto}, \mathtt{private} \rangle$, where

– `dabs` represents the non-pointer fields of the cell under the applied data abstraction,
– `pvars` is the set of (local to `th` or global) pointer variables that point to the cell,
– `reachfrom` is the set of (i) global pointer variables from which the cell represented by the tag is reachable via a (possibly empty) sequence of `next[1]` pointers, and (ii) observer registers $\mathtt{x}_i$ such that the cell is reachable from some cell whose data value equals that of $\mathtt{x}_i$,

- `reachto` is the corresponding information, but now considering cells that are reachable from the cell represented by the tag.
- `private` is `true` only if $\mathbb{c}$ is private to `th`.

Thus, the fragment contains both (i) *local* information about the cell's fields and variables that point to it, as well as (ii) *global* information, representing how each cell in the pair can reach to and be reached from (by following a chain of pointers) a small set of globally significant heap cells.

A set of fragments represents the set of heap structures in which each pair of pointer-connected nodes is represented by some fragment in the set. Put differently, a set of fragments describes the set of heaps that can be formed by "piecing together" pairs of pointer-connected nodes that are represented by some fragment in the set. This "piecing together" must be both locally consistent (appending only fragments that agree on their common node), and globally consistent (respecting the global reachability information). When applying fragment abstraction to skiplists, we use two types of fragments: *level 1-fragments* for nodes connected by a `next`[`1`]-pointer, and *higher level-fragments* for nodes connected by a higher level pointer. In other words, we abstract all levels higher than 2 by the abstract element `higher`. Thus, a pointer or non-pointer variable of form `v`[`k`], indexed by a level $\mathtt{k} \geq 2$, is abstracted to `v`[`higher`].

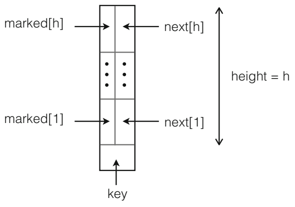


Fig. 3. A structure of a cell

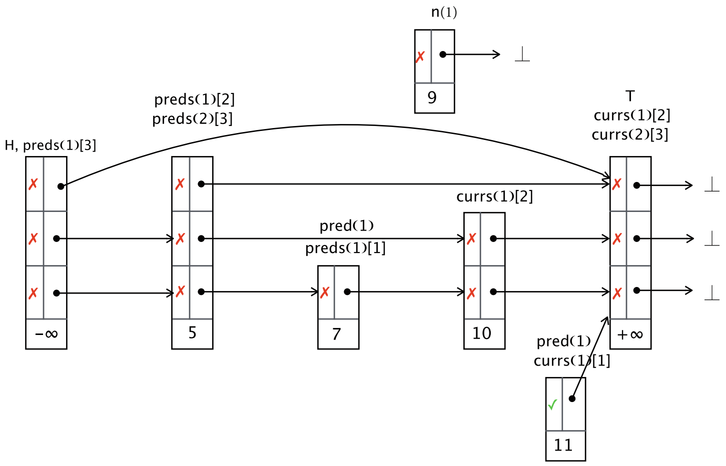


Fig. 4. A heap shape of a 3-level skiplist with two threads active

Let us illustrate how fragment abstraction applies to the skiplist algorithm. Figure 4 shows an example heap state of the skiplist algorithm with three levels.

Each heap cell is shown with the values of its fields as described in Fig. 3. In addition, each cell is labeled by the pointer variables that point to it; we use `preds`(`i`)[`k`] to denote the local variable `preds`[`k`] of thread $\mathtt{th_i}$, and the same for other local variables. In the heap state of Fig. 4, thread $\mathtt{th}_1$ is trying to add a new node of height 1 with key 9, and has reached line 8 of the `add` method. Thread $\mathtt{th}_2$ is trying to add a new node with key 20 and it has done its first iteration of the `for` loop in the `find` method. The variables `preds`(2)[3] and `curr`s(2)[3] have been assigned so that the new node (which has not yet been created) will be inserted between node 5 and the tail node. The observer is not shown, but the value of the observer register is 9; thus it currently tracks the `add` operation of $\mathtt{th}_1$.

Figure 5 illustrates how pairs of heap nodes can be represented by fragments. As a first example, in the view of thread $\mathtt{th}_1$, the two left-most cells in Fig. 4 are represented by the level 1-fragment $\mathtt{v}_1$ in Fig. 5. Here, the variable `preds`(1)[3] is represented by `preds`[`higher`]. The mapping π_1 represents the data abstraction of the `key` field, here saying that it is smaller than the value 9 of the observer register. The two left-most cells are also represented by a higher-level fragment, viz. v_8. The pair consisting of the two sentinel cells (with keys $-\infty$ and $+\infty$) is represented by the higher-level fragment v_9. In each fragment, the abstraction `dabs` of non-pointer fields are shown represented inside each tag of the fragment. The ϕ is shown as a label on the arrow between two tags. Above each tag is `pvars`. The first row under each tag is `reachfrom`, whereas the second row is `reachto`.

Figure 5 shows a set of fragments that is sufficient to represent the part of the heap that is accessible to $\mathtt{th}_1$ in the configuration in Fig. 4. There are 11 fragments, named $\mathtt{v}_1$, ..., $\mathtt{v}_{11}$. Two of these ($\mathtt{v}_6$, $\mathtt{v}_7$ and $\mathtt{v}_{11}$) consist of a tag

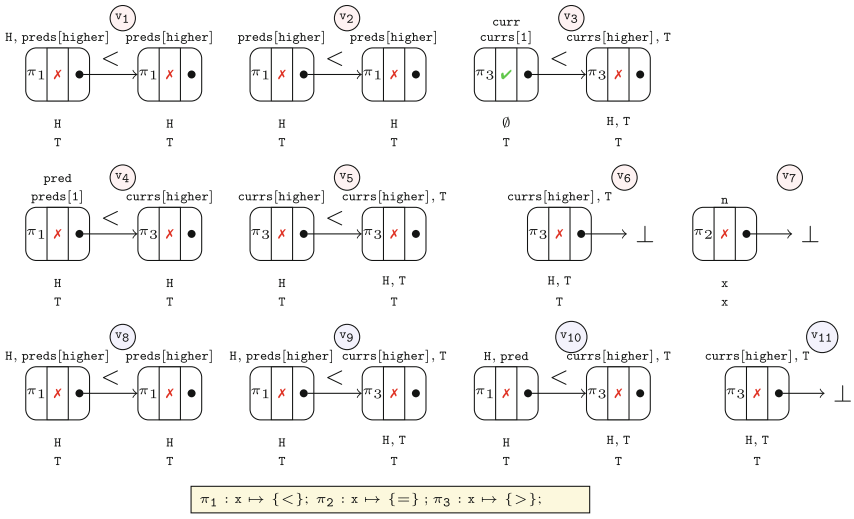


Fig. 5. Fragment abstraction of skiplist algorithm

that points to $\perp$. All other fragments consist of a pair of pointer-connected tags. The fragments $\mathtt{v}_1, \ldots, \mathtt{v}_6$ are level-1-fragments, whereas $\mathtt{v}_7, \ldots, \mathtt{v}_{11}$ are higher level-fragments. The `private` field of the input tag of $\mathtt{v}_7$ is `true`, whereas the `private` field of tags of other fragments are `false`.

To verify linearizability of the algorithm in Fig. 2, we must represent several key invariants of the heap. These include (among others):

1. the bottom-level list is strictly sorted in `key` order,
2. a higher-level pointer from a globally reachable node is a shortcut into the level-1 list, i.e., it points to a node that is reachable by a sequence of `next[1]` pointers,
3. all nodes which are unreachable from the head of the list are marked, and
4. the variable `pred` points to a cell whose `key` field is never larger than the input parameter of its `add` method.

Let us illustrate how such invariants are captured by our fragment abstraction. (1) All level-1 fragments are strictly sorted, implying that the bottom-level list is strictly sorted. (2) For each higher-level fragment `v`, if $\mathtt{H} \in$ `v.i.reachfrom` then also $\mathtt{H} \in$ `v.o.reachfrom`, implying (together with $\mathtt{v}.\phi = \{<\}$) that the cell represented by `v.o` it is reachable from that represented by `v.i` by a sequence of `next[1]`-pointers. (3) This is verified by inspecting each tag: $\mathtt{v}_3$ contains the only unreachable tag, and it is also marked. (4) The fragments express this property in the case where the value of `key` is the same as the value of the observer register `x`. Since the invariant holds for any value of `x`, this property is sufficiently represented for purposes of verification.

3 Concurrent Data Structure Implementations

In this section, we introduce our representation of concurrent data structure implementations, we define the correctness criterion of linearizability, we introduce observers and how to use them for specifying linearizability.

3.1 Concurrent Data Structure Implementations

We first introduce (sequential) data structures. A *data structure* `DS` is a pair $\langle \mathbb{D}, \mathbb{M} \rangle$, where $\mathbb{D}$ is a (possibly infinite) *data domain* and $\mathbb{M}$ is an alphabet of *method names*. An *operation op* is of the form $\mathtt{m}(d^{in}, d^{out})$, where $\mathtt{m} \in \mathbb{M}$ is a method name, and d^{in}, d^{out} are the *input* resp. *output* values, each of which is either in $\mathbb{D}$ or in some small finite domain $\mathbb{F}$, which includes the booleans. For some method names, the input or output value is absent from the operation. A *trace* of `DS` is a sequence of operations. The (sequential) semantics of a data structure `DS` is given by a set $[\![\mathtt{DS}]\!]$ of allowed traces. For example, a `Set` data structure has method names `add`, `remove`, and `contains`. An example of an allowed trace is `add(3, true) contains(4, false) contains(3, true) remove(3, true)`.

A *concurrent data structure implementation* operates on a shared state consisting of shared global variables and a shared heap. It assigns, to each method name, a method which performs operations on the shared state. It also comes with a method named `init`, which initializes its shared state.

A *heap (state)* $\mathcal{H}$ consists of a finite set $\mathbb{C}$ of cells, including the two special cells `null` and $\perp$ (dangling). Heap cells have a fixed set $\mathcal{F}$ of fields, namely non-pointer fields that assume values in $\mathbb{D}$ or $\mathbb{F}$, and possibly lock fields. We use the term $\mathbb{D}$-*field* for a non-pointer field that assumes values in $\mathbb{D}$, and the terms $\mathbb{F}$-*field* and *lock field* with analogous meaning. Furthermore, each cell has one or several named pointer fields. For instance, in data structure implementations based on singly-linked lists, each heap cell has a pointer field named `next`; in implementations based on skiplists there is an array of pointer fields named $\mathtt{next}[\mathtt{k}]$ where `k` ranges from 1 to a maximum level.

Each method declares local variables and a method body. The set of local variables includes the input parameter of the method and the program counter `pc`. A *local state* `loc` of a thread `th` defines the values of its local variables. The global variables can be accessed by all threads, whereas local variables can be accessed only by the thread which is invoking the corresponding method. Variables are either pointer variables (to heap cells), locks, or data variables assuming values in $\mathbb{D}$ or $\mathbb{F}$. We assume that all global variables are pointer variables. The body is built in the standard way from atomic commands, using standard control flow constructs (sequential composition, selection, and loop constructs). Atomic commands include assignments between variables, or fields of cells pointed to by a pointer variable. Method execution is terminated by executing a `return` command, which may return a value. The command `new Node()` allocates a new structure of type `Node` on the heap, and returns a reference to it. The compare-and-swap command $\mathtt{CAS}(\mathtt{a}, \mathtt{b}, \mathtt{c})$ atomically compares the values of `a` and `b`. If equal, it assigns the value of `c` to `a` and returns `true`, otherwise, it leaves `a` unchanged and returns `false`. We assume a memory management mechanism, which automatically collects garbage, and ensures that a new cell is fresh, i.e., has not been used before; this avoids the so-called ABA problem (e.g., [31]).

We define a *program* $\mathcal{P}$ (over a concurrent data structure) to consist of an arbitrary number of concurrently executing threads, each of which executes a method that performs an operation on the data structure. The shared state is initialized by the `init` method prior to the start of program execution. A *configuration* of a program $\mathcal{P}$ is a tuple $c_{\mathcal{P}} = \langle \mathtt{T}, \mathtt{LOC}, \mathcal{H} \rangle$ where `T` is a set of threads, $\mathcal{H}$ is a heap, and `LOC` maps each thread $\mathtt{th} \in \mathtt{T}$ to its local state $\mathtt{LOC}(\mathtt{th})$. We assume concurrent execution according to sequentially consistent memory model. The behavior of a thread `th` executing a method can be formalized as a transition relation $\rightarrow_{\mathtt{th}}$ on pairs $\langle \mathtt{loc}, \mathcal{H} \rangle$ consisting of a local state `loc` and a heap state $\mathcal{H}$. The behavior of a program $\mathcal{P}$ can be formalized by a transition relation $\rightarrow_{\mathcal{P}}$ on program configurations; each step corresponds to a move of a single thread. I.e., there is a transition of form $\langle \mathtt{T}, \mathtt{LOC}, \mathcal{H} \rangle \rightarrow_{\mathcal{P}} \langle \mathtt{T}, \mathtt{LOC}[\mathtt{th} \leftarrow \mathtt{loc}'], \mathcal{H}' \rangle$ whenever some thread $\mathtt{th} \in \mathtt{T}$ has a transition $\langle \mathtt{loc}, \mathcal{H} \rangle \rightarrow_{\mathtt{th}} \langle \mathtt{loc}', \mathcal{H}' \rangle$ with $\mathtt{LOC}(\mathtt{th}) = \mathtt{loc}$.

3.2 Linearizability

In a concurrent data structure implementation, we represent the calling of a method by a *call action* $\mathtt{call}_{\mathtt{o}}\ \mathtt{m}\left(d^{in}\right)$, and the return of a method by a *return action* $\mathtt{ret}_{\mathtt{o}}\ \mathtt{m}\left(d^{out}\right)$, where $\mathtt{o} \in \mathbb{N}$ is an *action identifier*, which links the call and return of each method invocation. A *history* h is a sequence of actions such that (i) different occurrences of return actions have different action identifiers, and (ii) for each return action a_2 in h there is a unique *matching* call action a_1 with the same action identifier and method name, which occurs before a_2 in h. A call action which does not match any return action in h is said to be *pending*. A history without pending call actions is said to be *complete*. A *completed extension* of h is a complete history h' obtained from h by appending (at the end) zero or more return actions that are matched by pending call actions in h, and thereafter removing the call actions that are still pending. For action identifiers $\mathtt{o}_1, \mathtt{o}_2$, we write $\mathtt{o}_1 \preceq_{\mathtt{h}} \mathtt{o}_2$ to denote that the return action with identifier $\mathtt{o}_1$ occurs before the call action with identifier $\mathtt{o}_2$ in h. A complete history is *sequential* if it is of the form $a_1 a_1' a_2 a_2' \cdots a_n a_n'$ where a_i' is the matching action of a_i for all $i : 1 \leq i \leq n$, i.e., each call action is immediately followed by its matching return action. We identify a sequential history of the above form with the corresponding trace $op_1 op_2 \cdots op_n$ where $op_i = \mathtt{m}(d_i^{in}, d_i^{out})$, $a_i = \mathtt{call}_{\mathtt{o}_i}\ \mathtt{m}\left(d_i^{in}\right)$, and $a_i = \mathtt{ret}_{\mathtt{o}_i}\ \mathtt{m}\left(d_i^{out}\right)$, i.e., we merge each call action together with the matching return action into one operation. A complete history h' is a *linearization* of h if (i) h' is a permutation of h, (ii) h' is sequential, and (iii) $\mathtt{o}_1 \preceq_{\mathtt{h}'} \mathtt{o}_2$ if $\mathtt{o}_1 \preceq_{\mathtt{h}} \mathtt{o}_2$ for each pair of action identifiers $\mathtt{o}_1$ and $\mathtt{o}_2$. A sequential history h' is *valid* wrt. $\mathtt{DS}$ if the corresponding trace is in $[\![\mathtt{DS}]\!]$. We say that h is *linearizable* wrt. $\mathtt{DS}$ if there is a completed extension of h, which has a linearization that is valid wrt. $\mathtt{DS}$. We say that a program $\mathcal{P}$ is linearizable wrt. $\mathtt{DS}$ if, in each possible execution, the sequence of call and return actions is *linearizable* wrt. $\mathtt{DS}$.

We specify linearizability using the technique of *observers* [1,3,7,9]. Depending on the data structure, we apply it in two different ways.

- For implementations of sets and priority queues, the user instruments each method so that it announces a corresponding operation precisely when the method executes its LP, either directly or with lightweight instrumentation using the technique of linearization policies [3]. We represent such announcements by labels on the program transition relation $\rightarrow_{\mathcal{P}}$, resulting in transitions of form $c_{\mathcal{P}} \xrightarrow{\mathtt{m}(d^{in}, d^{out})}_{\mathcal{P}} c_{\mathcal{P}}'$. Thereafter, an *observer* is constructed, which monitors the sequence of operations that is announced by the instrumentation; it reports (by moving to an accepting error location) whenever this sequence violates the (sequential) semantics of the data structure.
- For stacks and queues, we use a recent result [7,9] that the set of linearizable histories, i.e., sequences of call and return actions, can be exactly specified by an observer. Thus, linearizability can be specified without any user-supplied instrumentation, by using an observer which monitors the sequences of call and return actions and reports violations of linearizability.

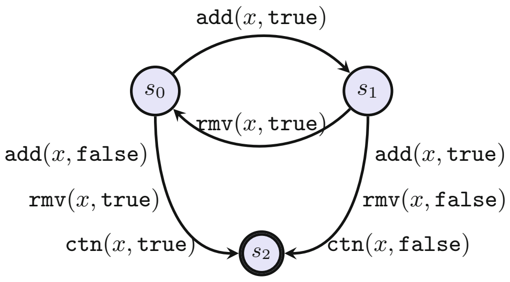


Fig. 6. Set observer.

Formally, an observer $\mathcal{O}$ is a tuple $\left\langle S^{\mathcal{O}}, s^{\mathcal{O}}_{\mathtt{init}}, \mathtt{X}^{\mathcal{O}}, \Delta^{\mathcal{O}}, s^{\mathcal{O}}_{\mathtt{acc}}\right\rangle$ where $S^{\mathcal{O}}$ is a finite set of *observer locations* including the *initial location* $s^{\mathcal{O}}_{\mathtt{init}}$ and the *accepting location* $s^{\mathcal{O}}_{\mathtt{acc}}$, a finite set $\mathtt{X}^{\mathcal{O}}$ of *registers*, and $\Delta^{\mathcal{O}}$ is a finite set of *transitions*. For observers that monitor sequences of operations, transitions are of the form $\left\langle s_1, \mathtt{m}(x^{in}, x^{out}), s_2\right\rangle$, where $\mathtt{m} \in \mathbb{M}$ is a method name and x^{in} and x^{out} are either registers or constants, i.e., transitions are labeled by operations whose input or output data may be parameterized on registers. The observer processes a sequence of operations one operation at a time. If there is a transition, whose label (after replacing registers by their values) matches the operation, such a transition is performed. If there is no such transition, the observer remains in its current location. The observer accepts a sequence if it can be processed in such a way that an accepting location is reached. The observer is defined in such a way that it accepts precisely those sequences that are *not* in $[\![\mathtt{DS}]\!]$. Figure 6 depicts an observer for the set data structure.

To check that no execution of the program announces a sequence of labels that can drive the observer to an accepting location, we form the cross-product $\mathcal{S} = \mathcal{P} \otimes \mathcal{O}$ of the program $\mathcal{P}$ and the observer $\mathcal{O}$, synchronizing on common transition labels. Thus, configurations of $\mathcal{S}$ are of the form $\langle c_{\mathcal{P}}, \langle s, \rho\rangle\rangle$, consisting of a program configuration $c_{\mathcal{P}}$, an observer location s, and an assignment ρ of values in $\mathbb{D}$ to the observer registers. Transitions of $\mathcal{S}$ are of the form $\langle c_{\mathcal{P}}, \langle s, \rho\rangle\rangle, \longrightarrow_{\mathcal{S}}, \langle c_{\mathcal{P}}', \langle s', \rho\rangle\rangle$, obtained from a transition $c_{\mathcal{P}} \xrightarrow{\lambda}_{\mathcal{P}} c_{\mathcal{P}}'$ of the program with some (possibly empty) label λ, where the observer makes a transition $s \xrightarrow{\lambda} s'$ if it can perform such a matching transition, otherwise $s' = s$. Note that the observer registers are not changed. We also add straightforward instrumentation to check that each method invocation announces exactly one operation, whose input and output values agree with the method's parameters and return value. This reduces the problem of checking linearizability to the problem of checking that in this cross-product, the observer cannot reach an accepting error location.

4 Verification Using Fragment Abstraction for Skiplists

In the previous section, we reduced the problem of verifying linearizability to the problem of verifying that, in any execution of the cross-product of a program and an observer, the observer cannot reach an accepting location. We perform this verification by computing a symbolic representation of an invariant that is satisfied by all reachable configurations of the cross-product, using an abstract interpretation-based fixpoint procedure, starting from a symbolic representation of the set of initial configurations, thereafter repeatedly performing symbolic postcondition computations that extend the symbolic representation by the effect of any execution step of the program, until convergence.

In Sect. 4.1, we define in more detail our symbolic representation for skiplists, focusing in particular on the use of fragment abstraction, and thereafter (in Sect. 4.2) describe the symbolic postcondition computation. Since singly-linked lists is a trivial special case of skiplists, we can use the relevant part of this technique also for programs based on singly-linked lists.

4.1 Symbolic Representation

This subsection contains a more detailed description of our symbolic representation for programs that operate on skiplists, which was introduced in Sect. 2. We first describe the data abstraction, thereafter the fragment abstraction, and finally their combination into a symbolic representation.

Data Abstraction. Our data abstraction is defined by assigning a abstract domain to each concrete domain of data values, as follows.

– For small concrete domains (including that of the program counter, and of the observer location), the abstract domain is the same as the concrete one.
– For locks, the abstract domain is $\{me, other, free\}$, meaning that the lock is held by the concerned thread, held by some other thread, or is free, respectively.
– For the concrete domain $\mathbb{D}$ of data values, the abstract domain is the set of mappings from observer registers and local variables ranging over $\mathbb{D}$ to subsets of $\{<, =, >\}$. An mapping in this abstract domain represents the set of data values $\mathtt{d}$ such that it maps each local variable and observer register with a value $\mathtt{d}' \in \mathbb{D}$ to a set which includes a relation $\sim$ such that $\mathtt{d} \sim \mathtt{d}'$.

Fragment Abstraction. Let us now define our fragment abstraction for skiplists. For presentation purposes, we assume that each heap cell has at most one $\mathbb{D}$-field, named `data`. For an observer register $\mathtt{x}_i$, let a $\mathtt{x}_i$-*cell* be a heap cell whose `data` field has the same value as $\mathtt{x}_i$.

Since the number of levels is unbounded, we define an abstraction for levels. Let $\mathtt{k}$ be a level. Define the abstraction of a pointer variable of form $\mathtt{p}[\mathtt{k}]$, denoted $\widehat{\mathtt{p}[\mathtt{k}]}$, to be $\mathtt{p}[1]$ if $\mathtt{k} = 1$, and to be $\mathtt{p}[\mathtt{higher}]$ if $\mathtt{k} \geq 2$. That is, this abstraction does not distinguish different higher levels.

A *tag* is a tuple $\mathtt{tag} = \langle \mathtt{dabs}, \mathtt{pvars}, \mathtt{reachfrom}, \mathtt{reachto}, \mathtt{private} \rangle$, where (i) `dabs` is a mapping from non-pointer fields to their corresponding abstract domains; if a non-pointer field is an array indexed by levels, then the abstract domain is that for single elements: e.g., the abstract domain for the array `marked` in Fig. 2 is simply the set of booleans, (ii) `pvars` is a set of abstracted pointer variables, (iii) `reachfrom` and `reachto` are sets of global pointer variables and observer registers, and (iv) `private` is a boolean value.

For a heap cell $\mathbb{c}$ that is accessible to thread `th` in a configuration $c_{\mathcal{S}}$, and a tag $\texttt{tag} = \langle \texttt{dabs}, \texttt{pvars}, \texttt{reachfrom}, \texttt{reachto}, \texttt{private} \rangle$, we let $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\texttt{th},\texttt{k}} \texttt{tag}$ denote that $\mathbb{c}$ satisfies the tag `tag` "at level `k`". More precisely, this means that

- `dabs` is an abstraction of the concrete values of the non-pointer fields of $\mathbb{c}$; for array fields `f` we use the concrete value `f`[`k`],
- `pvars` is the set of abstractions of pointer variables (global or local to `th`) that point to $\mathbb{c}$,
- `reachfrom` is the set of (i) abstractions of global pointer variables from which $\mathbb{c}$ is reachable via a (possibly empty) sequence of `next`[1] pointers, and (ii) observer registers $\texttt{x}_i$ such that $\mathbb{c}$ is reachable from some $\texttt{x}_i$-cell (via a sequence of `next`[1] pointers),
- `reachto` is the set of (i) abstractions of global pointer variables pointing to a cell that is reachable (via a sequence of `next`[1] pointers) from $\mathbb{c}$, and (ii) observer registers $\texttt{x}_i$ such that some $\texttt{x}_i$-cell is reachable from $\mathbb{c}$.
- `private` is `true` only if $\mathbb{c}$ is not accessible to any other thread than `th`.

Note that the global information represented by the fields `reachfrom` and `reachto` concerns *only* reachability via level-1 pointers.

A *skiplist fragment* `v` (or just fragment) is a triple of form $\langle \texttt{i}, \texttt{o}, \phi \rangle$, of form $\langle \texttt{i}, \texttt{null} \rangle$, or of form $\langle \texttt{i}, \perp \rangle$, where `i` and `o` are tags and ϕ is a subset of $\{<, =, >\}$. Each skiplist fragment additionally has a *type*, which is either *level-1* or *higher-level* (note that a level-1 fragment can otherwise be identical to a higher-level fragment). For a cell $\mathbb{c}$ which is accessible to thread `th`, and a fragment `v` of form $\langle \texttt{i}, \texttt{o}, \phi \rangle$, let $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\texttt{th},\texttt{k}} \texttt{v}$ denote that the `next`[k] field of $\mathbb{c}$ points to a cell $\mathbb{c}'$ such that $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\texttt{th},\texttt{k}} \texttt{i}$, and $\mathbb{c}' \lhd^{c_{\mathcal{S}}}_{\texttt{th},\texttt{k}} \texttt{o}$, and $\mathbb{c}.\texttt{data} \sim \mathbb{c}'.\texttt{data}$ for some $\sim \in \phi$. The definition of $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\texttt{th},\texttt{k}} \texttt{v}$ is adapted to fragments of form $\langle \texttt{i}, \texttt{null} \rangle$ and $\langle \texttt{i}, \perp \rangle$ in the obvious way. For a fragment $\texttt{v} = \langle \texttt{i}, \texttt{o}, \phi \rangle$, we often use `v.i` for `i` and `v.o` for `o`, etc.

Let V be a set of fragments. A global configuration $c_{\mathcal{S}}$ satisfies V wrp. to `th`, denoted $c_{\mathcal{S}} \models^{heap}_{\texttt{th}} V$, if

- for any cell $\mathbb{c}$ that is accessible to `th` (different from `null` and $\perp$), there is a level-1 fragment $\texttt{v} \in V$ such that $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\texttt{th},1} \texttt{v}$, and
- for all levels `k` from 2 up to the height of $\mathbb{c}$, there is a higher-level fragment $\texttt{v} \in V$ such that $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\texttt{th},\texttt{k}} \texttt{v}$.

Intuitively, a set of fragment represents the set of heap states, in which each pair of cells connected by a `next`[1] pointer is represented by a level-1 fragment, and each pair of cells connected by a `next`[k] pointer for $\texttt{k} \geq 2$ is represented by a higher-level fragment which represents array fields of cells at index `k`.

Symbolic Representation. We can now define our abstract symbolic representation.

Define a *local symbolic configuration* σ to be a mapping from local non-pointer variables (including the program counter) to their corresponding abstract domains. We let $c_{\mathcal{S}} \models^{loc}_{\mathtt{th}} \sigma$ denote that in the global configuration $c_{\mathcal{S}}$, the local configuration of thread `th` satisfies the local symbolic configuration σ, defined in the natural way. For a local symbolic configuration σ, an observer location s, a pair V of fragments and a thread `th`, we write $c_{\mathcal{S}} \models_{\mathtt{th}} \langle \sigma, s, V \rangle$ to denote that (i) $c_{\mathcal{S}} \models^{loc}_{\mathtt{th}} \sigma$, (ii) the observer is in location s, and (iii) $c_{\mathcal{S}} \models^{heap}_{\mathtt{th}} V$.

Definition 1. *A* symbolic representation Ψ *is a partial mapping from pairs of local symbolic configurations and observer locations to sets of fragments. A system configuration $c_{\mathcal{S}}$ satisfies a symbolic representation Ψ, denoted $c_{\mathcal{S}}$ sat Ψ, if for each thread* `th`*, the domain of Ψ contains a pair $\langle \sigma, s \rangle$ such that $c_{\mathcal{S}} \models_{\mathtt{th}} \langle \sigma, s, \Psi(\langle \sigma, s \rangle) \rangle$.*

4.2 Symbolic Postcondition Computation

The symbolic postcondition computation must ensure that the symbolic representation of the reachable configurations of a program is closed under execution of a statement by some thread. That is, given a symbolic representation Ψ, the symbolic postcondition operation must produce an extension Ψ' of Ψ, such that whenever $c_{\mathcal{S}}$ sat Ψ and $c_{\mathcal{S}} \rightarrow_{\mathcal{S}} c'_{\mathcal{S}}$ then $c_{\mathcal{S}}{}'$ sat Ψ'. Let `th` be an arbitrary thread. Then $c_{\mathcal{S}}$ sat Ψ means that $Dom(\Psi)$ contains some pair $\langle \sigma, s \rangle$ with $c_{\mathcal{S}} \models_{\mathtt{th}} \langle \sigma, s, \Psi(\langle \sigma, s \rangle) \rangle$. The symbolic postcondition computation must ensure that $Dom(\Psi')$ contains a pair $\langle \sigma', s' \rangle$ such that $c'_{\mathcal{S}} \models_{\mathtt{th}} \langle \sigma', s', \Psi'(\langle \sigma', s' \rangle) \rangle$. In the thread-modular approach, there are two cases to consider, depending on which thread causes the step from $c_{\mathcal{S}}$ to $c_{\mathcal{S}}{}'$.

- *Local Steps:* The step is caused by `th` itself executing a statement which may change its local state, the location of the observer, and the state of the heap. In this case, we first compute a local symbolic configuration σ', an observer location s', and a set V' of fragments such that $c'_{\mathcal{S}} \models_{\mathtt{th}} \langle \sigma', s', V' \rangle$, and then (if necessary) extend Ψ so that $\langle \sigma', s' \rangle \in Dom(\Psi)$ and $V' \subseteq \Psi(\langle \sigma', s' \rangle)$.
- *Interference Steps:* The step is caused by another thread $\mathtt{th}_2$, executing a statement which may change the location of the observer (to s') and the heap. By $c_{\mathcal{S}}$ sat Ψ there is a local symbolic configuration σ_2 with $\langle \sigma_2, s \rangle \in Dom(\Psi)$ such that $c_{\mathcal{S}} \models_{\mathtt{th}_2} \langle \sigma_2, s, \Psi(\langle \sigma_2, s \rangle) \rangle$. For any such σ_2 and statement of $\mathtt{th}_2$, we must compute a set V' of fragments such that the resulting configuration $c_{\mathcal{S}}{}'$ satisfies $c'_{\mathcal{S}} \models^{heap}_{\mathtt{th}} V'$ and ensure that $\langle \sigma, s' \rangle \in Dom(\Psi)$ and $V' \subseteq \Psi(\langle \sigma, s' \rangle)$. To do this, we first combine the local symbolic configurations σ and σ_2 and the sets of fragments $\Psi(\langle \sigma, s \rangle)$ and $\Psi(\langle \sigma_2, s \rangle)$, using an operation called *intersection*, into a joint local symbolic configuration of `th` and $\mathtt{th}_2$ and a set $V_{1,2}$ of fragments that represents the cells accessible to either `th` or $\mathtt{th}_2$. We thereafter symbolically compute the postcondition of the statement executed by $\mathtt{th}_2$, in the same was as for local steps, and finally project the set of resulting fragments back onto `th` to obtain V'.

In the following, we first describe the symbolic postcondition computation for local steps, and thereafter the intersection operation.

Symbolic Postcondition Computation for Local Steps. Let th be an arbitrary thread, assume that $\langle \sigma, s \rangle \in Dom(\Psi)$, and let $V = \Psi(\langle \sigma, s \rangle)$ For each statement that th can execute in a configuration $c_{\mathcal{S}}$ with $c_{\mathcal{S}} \models_{\mathtt{th}} \langle \sigma, s, V \rangle$, we must compute a local symbolic configuration σ', a new observer location s' and a set V' of fragments such that the resulting configuration $c_{\mathcal{S}}{}'$ satisfies $c'_{\mathcal{S}} \models_{\mathtt{th}} \langle \sigma', s', V' \rangle$. This computation is done differently for each statement. For statements that do not affect the heap or pointer variables, this computation is standard, and affects only the local symbolic configuration, the observer location, and the dabs component of tags. We therefore here describe how to compute the effect of statements that update pointer variables or pointer fields of heap cells, since these are the most interesting cases. In this computation, the set V' is constructed in two steps: (1) First, the level-1 fragments of V' are computed, based on the level-1 fragments in V. (2) Thereafter, the higher-level fragments of V' are computed, based on the higher-level fragments in V and how fragments in V are transformed when entered in to V'. We first describe the construction of level-1 fragments, and thereafter the construction of higher-level fragments.

Construction of Level-1 Fragments. Let us first intuitively introduce techniques used for constructing the level-1 fragments of V'. Consider a statement of form g := p, which assigns the value of a local pointer variable p to a global pointer variable g. The set V' of fragments is obtained by modifying fragments in V to reflect the effect of the assignment. For any tag in a fragment, the dabs field is not affected. The pvars field is updated to contain the variable g if and only if it contained the variable p before the statement. The difficulty is to update the reachability information represented by the fields reachfrom and reachto, and in particular to determine whether g should be in such a set after the statement (note that if p were a global variable, then the corresponding reachability information for p would be in the fields reachfrom and reachto, and the update would be simple, reflecting that g and p become aliases). In order to construct V' with sufficient precision, we therefore investigate whether the set of fragments V allows to form a heap in which a p-cell can reach or be reached from (by a sequence of next[1] pointers) a particular tag of a fragment. We also investigate whether a heap can be formed in which a *p*-cell can *not* reach or be reached from a particular tag. For each such successful investigation, the set V' will contain a level-1 fragment with corresponding contents of its reachto and reachfrom fields.

The postcondition computation performs this investigation by computing a set of transitive closure-like relations between level-1 fragments, which represent reachability via sequences of next[1] pointers (since only these are relevant for the reachfrom and reachto fields). First, say that two tags tag and tag′ are *consistent* (wrp. to a set of fragments V) if the concretizations of their dabs-fields overlap, and if the other fields pvars, reachfrom, reachto, and private) agree. Thus, tag and tag′ are consistent if there can exist a cell $\mathbb{c}$ accessible to `th` in some heap, with $\mathbb{c} \lhd^{cs}_{\mathtt{th}}$ `tag` and $\mathbb{c} \lhd^{cs}_{\mathtt{th}}$ `tag'`. Next, for two level-1 fragments $\mathtt{v}_1$ and $\mathtt{v}_2$ in a set V of fragments,

- let $\mathtt{v}_1 \hookrightarrow_V \mathtt{v}_2$ denote that $\mathtt{v}_1.\mathtt{o}$ and $\mathtt{v}_2.\mathtt{i}$ are consistent, and
- let $\mathtt{v}_1 \leftrightarrow_V \mathtt{v}_2$ denote that $\mathtt{v}_1.\mathtt{o} = \mathtt{v}_2.\mathtt{o}$ are consistent, and that either $\mathtt{v}_1.\mathtt{i}.\mathtt{pvars} \cap \mathtt{v}_2.\mathtt{i}.\mathtt{pvars} = \emptyset$ or the global variables in $\mathtt{v}_1.\mathtt{i}.\mathtt{reachfrom}$ are disjoint from those in $\mathtt{v}_2.\mathtt{i}.\mathtt{reachfrom}$.

Intuitively, $\mathtt{v}_1 \hookrightarrow_V \mathtt{v}_2$ denotes that it is possible that $\mathbb{c}_1.\mathtt{next}[1] = \mathbb{c}_2$ for some cells with $\mathbb{c}_1 \lhd^{cs}_{\mathtt{th},1} \mathtt{v}_1$ and $\mathbb{c}_2 \lhd^{cs}_{\mathtt{th},1} \mathtt{v}_2$. Intuitively, $\mathtt{v}_1 \leftrightarrow_V \mathtt{v}_2$ denotes that it is possible that $\mathbb{c}_1.\mathtt{next}[1] = \mathbb{c}_2.\mathtt{next}[1]$ for different cells $\mathbb{c}_1$ and $\mathbb{c}_2$ with $\mathbb{c}_1 \lhd^{cs}_{\mathtt{th},1} \mathtt{v}_1$ and $\mathbb{c}_2 \lhd^{cs}_{\mathtt{th},1} \mathtt{v}_2$ (Note that these definitions also work for fragments containing `null` or $\perp$). We use these relations to define the following derived relations on level-1 fragments:

- $\overset{+}{\hookrightarrow}_V$ denotes the transitive closure, and $\overset{*}{\hookrightarrow}_V$ the reflexive transitive closure, of $\hookrightarrow_V$,
- $\mathtt{v}_1 \overset{**}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1, \mathtt{v}'_2 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}'_2$ where $\mathtt{v}_1 \overset{*}{\hookrightarrow}_V \mathtt{v}'_1$ and $\mathtt{v}_2 \overset{*}{\hookrightarrow}_V \mathtt{v}'_2$,
- $\mathtt{v}_1 \overset{*+}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1, \mathtt{v}'_2 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}'_2$ where $\mathtt{v}_1 \overset{*}{\hookrightarrow}_V \mathtt{v}'_1$ and $\mathtt{v}_2 \overset{+}{\hookrightarrow}_V \mathtt{v}'_2$,
- $\mathtt{v}_1 \overset{*\circ}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}_2$ where $\mathtt{v}_1 \overset{*}{\hookrightarrow}_V \mathtt{v}'_1$,
- $\mathtt{v}_1 \overset{++}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1, \mathtt{v}'_2 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}'_2$ where $\mathtt{v}_1 \overset{+}{\hookrightarrow}_V \mathtt{v}'_1$ and $\mathtt{v}_2 \overset{+}{\hookrightarrow}_V \mathtt{v}'_2$,
- $\mathtt{v}_1 \overset{+\circ}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}_2$ where $\mathtt{v}_1 \overset{+}{\hookrightarrow}_V \mathtt{v}'_1$.

We sometimes use, e.g., $\mathtt{v}_2 \overset{+*}{\leftrightarrow}_V \mathtt{v}_1$ for $\mathtt{v}_1 \overset{*+}{\leftrightarrow}_V \mathtt{v}_2$. We say that $\mathtt{v}_1$ and $\mathtt{v}_2$ are *compatible* if $\mathtt{v}_x \overset{*}{\hookrightarrow} \mathtt{v}_y$, or $\mathtt{v}_y \overset{*}{\hookrightarrow} \mathtt{v}_x$, or $\mathtt{v}_x \overset{**}{\leftrightarrow} \mathtt{v}_y$. Intuitively, if $\mathtt{v}_1$ and $\mathtt{v}_2$ are satisfied by two cells in the same heap state, then they must be compatible.

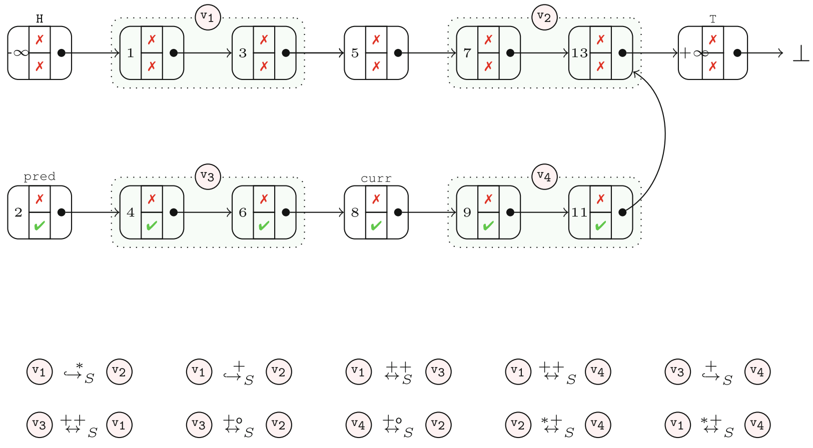

H
v1
v2
T
-∞
1
3
5
7
13
+∞
⊥
pred
v3
curr
v4
2
4
6
8
9
11
$v_1 \overset{*}{\hookrightarrow}_S v_2$
$v_1 \overset{+}{\hookrightarrow}_S v_2$
$v_1 \overset{++}{\leftrightarrow}_S v_3$
$v_1 \overset{++}{\leftrightarrow}_S v_4$
$v_3 \overset{+}{\hookrightarrow}_S v_4$
$v_3 \overset{++}{\leftrightarrow}_S v_1$
$v_3 \overset{+\circ}{\leftrightarrow}_S v_2$
$v_4 \overset{+\circ}{\leftrightarrow}_S v_2$
$v_2 \overset{*+}{\leftrightarrow}_S v_4$
$v_1 \overset{*+}{\leftrightarrow}_S v_4$

Fig. 7. Illustration of some transitive closure-like relations between fragments

Figure 7 illustrates the above relations for a heap state with 13 heap cells. The figure depicts, in green, four pairs of heap cells connected by a $\texttt{next}[1]$ pointer, which satisfy the four fragments $\mathtt{v}_1$, $\mathtt{v}_2$, $\mathtt{v}_3$, and $\mathtt{v}_4$, respectively. At the bottom are depicted the transitive-closure like relations that hold between these fragments.

We can now describe the symbolic postcondition computation for statements that affect pointer variables or fields. This is a case analysis, and for space reasons we only include some representative cases.

First, consider a statement of form $\mathtt{x} := \mathtt{y}$, where $\mathtt{x}$ and $\mathtt{y}$ are local (to thread $\mathtt{th}$) or global pointer variables. We must compute a set V' of fragments which are satisfied by the configuration after the statement. We first compute the level-1-fragments in V' as follows (higher-level fragments will be computed later). We observe that for any cell $\mathbb{c}$ which is accessible to $\mathtt{th}$ after the statement, there must be some level-1 fragment $\mathtt{v}'$ in V' with $\mathbb{c} \lhd^{cs}_{\mathtt{th},1} \mathtt{v}'$. By assumption, $\mathbb{c}$ satisfies some fragment $\mathtt{v}$ in V before the statement, and is in the same heap state as the cell pointed to by $\mathtt{y}$. This implies that $\mathtt{v}$ must be compatible with some fragment $\mathtt{v}_y \in V$ such that $\widehat{\mathtt{y}} \in \mathtt{v}_y.\mathtt{i.pvars}$ (recall that $\widehat{\mathtt{y}}$ is the abstraction of $\mathtt{y}$, which in the case that $\mathtt{y}$ is an array element maps higher level indices to that abstract index $\mathtt{higher}$). This means that we can make a case analysis on the possible relationships between $\mathtt{v}$ and any such $\mathtt{v}_y$. Thus, for each fragment $\mathtt{v}_y \in V$ such that $\widehat{\mathtt{y}} \in \mathtt{v}_y.\mathtt{i.pvars}$ we let V' contain the fragments obtained by any of the following transformations on any fragment in V.

1. First, for the fragment $\mathtt{v}_y$ itself, we let V' contain $\mathtt{v}'_y$, which is the same as $\mathtt{v}_y$, except that
   - $\mathtt{v}'_y.\mathtt{i.pvars} = \mathtt{v}_y.\mathtt{i.pvars} \cup \{\widehat{\mathtt{x}}\}$ and $\mathtt{v}'_y.\mathtt{o.pvars} = \mathtt{v.o.pvars} \setminus \{\widehat{\mathtt{x}}\}$

   and furthermore, if $\mathtt{x}$ is a global variable, then
   - $\mathtt{v}'_y.\mathtt{i.reachto} = \mathtt{v}_y.\mathtt{i.reachto} \cup \{\widehat{\mathtt{x}}\}$ and $\mathtt{v}'_y.\mathtt{i.reachfrom} = \mathtt{v}_y.\mathtt{i.reachfrom} \cup \{\widehat{\mathtt{x}}\}$,
   - $\mathtt{v}'_y.\mathtt{o.reachfrom} = \mathtt{v}_y.\mathtt{o.reachfrom} \cup \{\widehat{\mathtt{x}}\}$ and $\mathtt{v}'_y.\mathtt{o.reachto} = \mathtt{v}_y.\mathtt{o.reachto} \setminus \{\widehat{\mathtt{x}}\}$.
2. for each $\mathtt{v}$ with $\mathtt{v} \hookrightarrow_V \mathtt{v}_y$, let V' contain $\mathtt{v}'$ which is the same as $\mathtt{v}$ except that
   - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v.i.pvars} \setminus \{\widehat{\mathtt{x}}\}$,
   - $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v.o.pvars} \cup \{\widehat{\mathtt{x}}\}$,
   - $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v.i.reachfrom} \setminus \{\widehat{\mathtt{x}}\}$ if $\mathtt{x}$ is a global variable,
   - $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v.i.reachto} \cup \{\widehat{\mathtt{x}}\}$ if $\mathtt{x}$ is a global variable,
   - $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v.o.reachfrom} \cup \{\widehat{\mathtt{x}}\}$ if $\mathtt{x}$ is a global variable,
   - $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v.o.reachto} \cup \{\widehat{\mathtt{x}}\}$ if $\mathtt{x}$ is a global variable,
3. We perform analogous inclusions for fragments $\mathtt{v}$ with $\mathtt{v} \overset{+}{\hookrightarrow}_V \mathtt{v}_y$, $\mathtt{v}_y \overset{*}{\hookrightarrow}_V \mathtt{v}$, $\mathtt{v}_y \overset{*+}{\leftrightarrow}_V \mathtt{v}$, and $\mathtt{v}_y \overset{*\circ}{\leftrightarrow}_V \mathtt{v}$. Here, we show only the case of $\mathtt{v}_y \overset{*+}{\leftrightarrow}_V \mathtt{v}$, in which case we let V' contain $\mathtt{v}'$ which is the same as $\mathtt{v}$ except that $\widehat{\mathtt{x}}$ is removed from the sets $\mathtt{v}'.\mathtt{i.pvars}$, $\mathtt{v}'.\mathtt{o.pvars}$, $\mathtt{v}'.\mathtt{i.reachfrom}$, $\mathtt{v}'.\mathtt{i.reachto}$, $\mathtt{v}'.\mathtt{o.reachfrom}$, and $\mathtt{v}'.\mathtt{o.reachto}$.

The statement $\mathtt{x} := \mathtt{y.next}[1]$ is handled rather similarly to the case $\mathtt{x} := \mathtt{y}$. Let us therefore describe the postcondition computation for statements of the form $\mathtt{x.next}[1] := \mathtt{y}$. This is the most difficult statement, since it is a destructive update of the heap. It affects reachability relations for both $\mathtt{x}$ and $\mathtt{y}$. The postcondition computation makes a case analysis on how a fragment in V is related to some pair of compatible fragments $\mathtt{v}_x$, $\mathtt{v}_y$ in V such that $\widehat{\mathtt{x}} \in \mathtt{v}_x.\mathtt{i.pvars}$, $\widehat{\mathtt{y}} \in \mathtt{v}_y.\mathtt{i.pvars}$. Thus, for each pair of compatible fragments $\mathtt{v}_x$, $\mathtt{v}_y$ in V such that $\widehat{\mathtt{x}} \in \mathtt{v}_x.\mathtt{i.pvars}$ and $\widehat{\mathtt{y}} \in \mathtt{v}_y.\mathtt{i.pvars}$, it is first checked whether the statement may form a cycle in the heap. This may happen if $\mathtt{v}_y \overset{*}{\hookrightarrow}_V \mathtt{v}_x$, in which case the postcondition computation reports a potential cycle. Otherwise, V' consists of

1. the fragment $\mathtt{v}_{new}$, representing the new pair of neighbours formed by the statement, of form $\mathtt{v}_{new} = \langle \mathtt{i}, \mathtt{o}, \phi \rangle$, such that $\mathtt{v}_{new}.\mathtt{i.tag} = \mathtt{v}_x.\mathtt{i.tag}$ and $\mathtt{v}_{new}.\mathtt{o.tag} = \mathtt{v}_y.\mathtt{i.tag}$ except that $\mathtt{v}_{new}.\mathtt{o.reachfrom} = \mathtt{v}_y.\mathtt{i.reachfrom} \cup \mathtt{v}_x.\mathtt{i.reachfrom}$ and $\mathtt{v}_{new}.\mathtt{i.reachto} = \mathtt{v}_y.\mathtt{i.reachto} \cup \mathtt{v}_x.\mathtt{i.pvars}$; the constraint represent by $\mathtt{v}_{new}.\phi$ is obtained from the constraints represented by the data abstractions of $\mathtt{v}_x.\mathtt{i}$ and $\mathtt{v}_y.\mathtt{i}$, as well as the possible transitive closure-relations between $\mathtt{v}_x$ and $\mathtt{v}_y$, some of which imply that the data fields of $\mathtt{v}_x$ and $\mathtt{v}_y$ are ordered, and
2. all possible fragments that can result from a transformation of some fragment $\mathtt{v} \in V$. This is done by an exhaustive case analysis on the possible relationships between $\mathtt{v}$, $\mathtt{v}_x$ and $\mathtt{v}_y$. Let us consider an interesting case, in which $\mathtt{v}_x \overset{*}{\hookrightarrow}_V \mathtt{v}$ and either $\mathtt{v} \overset{+}{\hookrightarrow}_V \mathtt{v}_y$ or $\mathtt{v}_y \overset{*+}{\leftrightarrow} \mathtt{v}$. In this case,
   - for each subset $\mathtt{regset}$ of the observer registers in $\mathtt{v.i.reachfrom} \cap \mathtt{v}_x.\mathtt{i.reachfrom}$, and for each subset $\mathtt{regset}'$ of the set of observer registers in $\mathtt{v.o.reachfrom} \cap \mathtt{v}_x.\mathtt{i.reachfrom}$, we let V' contain a fragment $\mathtt{v}'$ which is the same as $\mathtt{v}$ except that $\mathtt{v}'.\mathtt{i.reachfrom} = (\mathtt{v.i.reachfrom} \setminus \mathtt{v_x.i.reachfrom}) \cup \mathtt{regset}$ and $\mathtt{v}'.\mathtt{o.reachfrom} = (\mathtt{v.o.reachfrom} \setminus \mathtt{v}_x.\mathtt{i.reachfrom}) \cup \mathtt{regset}'$. An intuitive explanation for the rule for $\mathtt{v}'.\mathtt{i.reachfrom}$ is that the global variables that can reach $\mathtt{v}_x.\mathtt{i}$ should clearly be removed from $\mathtt{v}'.\mathtt{i.reachfrom}$ since $\mathtt{v}_x \overset{*}{\hookrightarrow}_V \mathtt{v}'$ is false after the statement. However, for an observer register $\mathtt{x}_i$, an $\mathtt{x}_i$-cell can still reach $\mathtt{v}'.\mathtt{i}$, if there are two $\mathtt{x}_i$-cells, one which reaches $\mathtt{v}_x.\mathtt{i}$ and another which reaches $\mathtt{v}'.\mathtt{i}$; we cannot precisely determine for which $\mathtt{x}_i$ this may be the case, except that any such $\mathtt{x}_i$ must be in $\mathtt{v.i.reachfrom} \cap \mathtt{v}_x.\mathtt{i.reachfrom}$. The intuition for the rule for $\mathtt{v}'.\mathtt{o.reachfrom}$ is analogous.

Construction of Higher-Level Fragments. Based on the above construction of level-1 fragments, the set of higher-level fragments in V' is obtained as follows. For each higher level-fragment $\mathtt{v} \in V$, let $\mathtt{v}_1$ and $\mathtt{v}_2$ be level 1-fragments such that $\mathtt{v}_1.\mathtt{i.tag} = \mathtt{v.i.tag}$ and $\mathtt{v}_2.\mathtt{i.tag} = \mathtt{v.o.tag}$. For any fragments $\mathtt{v}_1'$ and $\mathtt{v}_2'$ that are derived from $\mathtt{v}_1$ and $\mathtt{v}_2$, respectively, V' contains a higher-level fragment $\mathtt{v}'$ which is the same as $\mathtt{v}$ except that (i) $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v}_1'.\mathtt{i.pvars}$ and $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v}_2'.\mathtt{i.pvars}$, (ii) $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v}_1'.\mathtt{i.reachfrom}$ and $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v}_2'.\mathtt{i.reachfrom}$, and (iii) $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v}_1'.\mathtt{i.reachto}$ and $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v}_2'.\mathtt{i.reachto}$. In addition, a statement of form $\mathtt{x.next[k]} := \mathtt{y}$ for $\mathtt{k} \geq 2$ creates a new fragment. The formation of this fragment is simpler than for the statement $\mathtt{x.next[1]} := \mathtt{y}$, since reachability via $\mathtt{next[1]}$-pointers is preserved.

Symbolic Postcondition Computation for Interference Steps. Here, the key step is the *intersection* operation, which takes two sets of fragments V_1 and V_2, and produces a set of joint fragments $V_{1,2}$, such that $c_{\mathcal{S}} \models^{heap}_{\mathtt{th}_1,\mathtt{th}_2} V_{1,2}$ for any configuration such that $c_{\mathcal{S}} \models^{heap}_{\mathtt{th}_i} V_i$ for $i = 1, 2$ (here $\models^{heap}_{\mathtt{th}_1,\mathtt{th}_2}$ is defined in the natural way). This means that for each heap cell accessible to either $\mathtt{th}_1$ or $\mathtt{th}_2$, the set $V_{1,2}$ contains a fragment v with $\mathbb{c} \lhd^{cs}_{\{\mathtt{th}_1,\mathtt{th}_2\},\mathtt{k}} \mathtt{v}$ for each k which is at most the height of $\mathbb{c}$ (generalizing the notation $\lhd^{cs}_{\mathtt{th},\mathtt{k}}$ to several threads). Note that a joint fragment represents local pointer variables of both $\mathtt{th}_1$ and $\mathtt{th}_2$. In order to distinguish between local variables of $\mathtt{th}_1$ and $\mathtt{th}_2$, we use x[i] to denote a local variable x of thread $\mathtt{th}_i$. Here, we describe the intersection operation for level-1 fragments. The intersection operation is analogous for higher-level fragments.

For a fragment v, define v.i.greachfrom as the set of global variables in v.i.reachfrom. Define v.i.greachto, v.o.greachfrom, v.o.greachto, v.i.gpvars, and v.o.gpvars analogously. Define v.i.gtag as the tuple $\langle$v.i.dabs, v.i.gpvars, v.i.greachfrom, v.i.greachto$\rangle$, and define v.o.gtag analogously. We must distinguish the following possibilities.

- If $\mathbb{c}$ is accessible to both $\mathtt{th}_1$ and $\mathtt{th}_2$, then there are fragments $\mathtt{v}_1 \in V_1$ and $\mathtt{v}_2 \in V_2$ such that $\mathbb{c} \lhd^{cs}_{\mathtt{th}_1,1} \mathtt{v}_1$ and $\mathbb{c} \lhd^{cs}_{\mathtt{th}_2,1} \mathtt{v}_2$. This can happen only if $\mathtt{v}_1$.i.gtag = $\mathtt{v}_2$.i.gtag, and $\mathtt{v}_1$.o.gtag = $\mathtt{v}_2$.o.gtag, and $\mathtt{v}_1$.i.private = $\mathtt{v}_2$.i.private = false. Thus, for any such pair of fragments $\mathtt{v}_1 \in V_1$ and $\mathtt{v}_2 \in V_2$, we let $V_{1,2}$ contain a fragment $\mathtt{v}_{12}$ which is identical to $\mathtt{v}_1$ except that
  - $\mathtt{v}_{12}$.i.pvars = $\mathtt{v}_1$.i.pvars $\cup$ $\mathtt{v}_2$.i.pvars,
  - $\mathtt{v}_{12}$.o.pvars = $\mathtt{v}_1$.o.pvars $\cup$ $\mathtt{v}_2$.o.pvars,
  - $\mathtt{v}_{12}$.i.reachfrom = $\mathtt{v}_1$.i.reachfrom $\cup$ $\mathtt{v}_2$.i.reachfrom, and
  - $\mathtt{v}_{12}$.o.reachfrom = $\mathtt{v}_1$.o.reachfrom $\cup$ $\mathtt{v}_2$.o.reachfrom.
- If $\mathbb{c}$ is accessible to $\mathtt{th}_1$, but not to $\mathtt{th}_2$, and $\mathbb{c}$.next[1] is accessible also to $\mathtt{th}_2$, then there are fragments $\mathtt{v}_1 \in V_1$ and $\mathtt{v}_2 \in V_2$ such that $\mathbb{c} \lhd^{cs}_{\mathtt{th}_1,1} \mathtt{v}_1$ and $\mathbb{c}$.next[1] $\lhd^{cs}_{\mathtt{th}_2,1} \mathtt{v}_2$.o. This can happen only if $\mathtt{v}_1$.i.greachfrom = $\emptyset$, and $\mathtt{v}_1$.o.gtag = $\mathtt{v}_2$.o.gtag, and $\mathtt{v}_1$.o.private = $\mathtt{v}_2$.o.private = false. Thus, for any such pair of fragments $\mathtt{v}_1 \in V_1$ and $\mathtt{v}_2 \in V_2$, we let $V_{1,2}$ contain a fragment $\mathtt{v}'_1$ which is identical to $\mathtt{v}_1$ except that
  - $\mathtt{v}'_1$.o.pvars = $\mathtt{v}_1$.o.pvars $\cup$ $\mathtt{v}_2$.o.pvars, and
  - $\mathtt{v}'_1$.o.reachfrom = $\mathtt{v}_1$.o.reachfrom $\cup$ $\mathtt{v}_2$.o.reachfrom.
- If neither $\mathbb{c}$ nor $\mathbb{c}$.next[1] is accessible $\mathtt{th}_2$, then there is a fragment $\mathtt{v}_1 \in V_1$ such that $\mathbb{c} \lhd^{cs}_{\mathtt{th}_1,1} \mathtt{v}_1$. This can happen only if $\mathtt{v}_1$.o.greachfrom = $\emptyset$, in which case we let $V_{1,2}$ contain the fragment $\mathtt{v}_1$.
- For each of the two last cases, there is also a symmetric case with the roles of $\mathtt{th}_1$ and $\mathtt{th}_2$ reversed.

5 Arrays of Singly-Linked Lists with Timestamps

In this section, we show how to apply fragment abstraction to concurrent programs that operate on a shared heap which represents an array of singly linked lists. We use this abstraction to provide the first automated verification of linearizability for the Timedstamped stack and Timestamped queue algorithms of [12] as reported in Sect. 6.

```
struct Node {
          int data;
          Timestamp ts;
          Node* next;
          boolean mark;
       }
```

```
init() :
Node* pools[maxThreads];
for(int i=1; i<=maxThreads; i++)
pools[i].next = null;
```

```
void push(int d):
1  Node* new := new Node(d,-1,null,false);
2  new.next = pools[myID];
3  pools[myID] = new;
4  Timestamp t = new Timestamp();
5  new.ts = t;
6  Node* next = new.next;
7  while (next.next != next & !next.mark)
8    next = next.next;
9  new.next = next;
10 return new;
```

```
int pop():
1  boolean success = false;
2  int maxTS = -1;
3  Node* youngest, myTop, n = null;
4  while (!success)
5    int k;
6    for(int i=1; i<=maxThreads; i++)
7      n = pools[i];
8      while (n.mark & n.next != n) n = n.next;
9      if(maxTS < n.ts)
10       maxTS = n.ts;
11       youngest = n;
12       k = i; myTop = pools[k];
13   if (youngest != null)
14     success = CAS(youngest.mark,false,true);
15       if (success)
16         CAS(pools[k], myTop, youngest);
17         if (myTop != youngest);
18           myTop.next = youngest;
19         pools[k].next = youngest.next;
20         Node* next=youngest.next
21         while (next.next != next & next.mark);
22           next = next.next;
23         youngest.next = next;
24 return youngest.data;
```

Fig. 8. Description of the Timestamped stack algorithm, with some simplifications.

Figure 8 shows a simplified version of the Timestamped Stack (TS stack) of [12], where we have omitted the check for emptiness in the `pop` method, and the optimization using `push-pop` elimination. These features are included in the full version of the algorithm, that we have verified automatically.

The algorithm uses an array of singly-linked lists (SLLs), one for each thread, accessed via the thread-indexed array `pools`[`maxThreads`] of pointers to the first cell of each list. The `init` method initializes each of these pointers to `null`. Each list cell contains a data value, a timestamp value, a `next` pointer, and a boolean flag `mark` which indicates whether the node is logically removed from the stack. Each thread pushes elements only to "its own" list, but can pop elements from any list.

A `push` method for inserting a data element `d` works as follows: first, a new cell with element `d` and minimal timestamp -1 is inserted at the beginning of the list indexed by the calling thread (line 1–3). After that, a new timestamp is created and assigned (via the variable `t`) to the `ts` field of the inserted cell (line 4–5). Finally, the method unlinks (i.e., physically removes) all cells that are reachable (through a sequence of `next` pointers) from the inserted cell and whose `mark` field is `true`; these cells are already logically removed. This is done by redirecting the `next` pointer of the inserted cell to the first cell with a `false` `mark` field, which is reachable from the inserted cell.

A `pop` method first traverses all lists, finding in each list the first cell whose `mark` field is `false` (line 8), and letting the variable `youngest` point to the most recent such cell (i.e., with the largest timestamp) (line 1–11). A compare-and-swap (CAS) is used to set the `mark` field of this youngest cell to `true`, thereby logically removing it. This procedure will restart if the CAS fails. After the youngest cell has been removed, the method will unlink all cells, whose `mark` field is `true`, that appear before (line 17–19) or after (line 20–23) the removed cell. Finally, the method returns the `data` value of the removed cell.

Fragment Abstraction. In our verification, we establish that the TS stack algorithm of Fig. 8 is correct in the sense that it is a linearizable implementation of a stack data structure. For stacks and queues, we specify linearizability by observers that synchronize on call and return actions of methods, as shown by [7]; this is done without any user-supplied annotation, hence the verification is fully automated.

The verification is performed analogously as for skiplists, as described in Sect. 4. Here we show how fragment abstraction is used for arrays of singly-linked lists. Figure 9 shows an example heap state of TS stack. The heap consists of a set of singly linked lists (SLLs), each of which is accessed from a pointer in the array `pools`[`maxThreads`] in a configuration when it is accessed concurrently by three threads $\mathtt{th}_1$, $\mathtt{th}_2$, and $\mathtt{th}_3$. The heap consists of three SLLs accessed from the three pointers `pools`[1], `pools`[2], and `pools`[3] respectively. Each heap cell is shown with the values of its fields, using the layout shown to the right in Fig. 9. In addition, each cell is labeled by the pointer variables that point to it. We use `lvar`(`i`) to denote the local variable `lvar` of thread $\mathtt{th}_\mathtt{i}$.

In the heap state of Fig. 9, thread $\mathtt{th}_1$ is trying to push a new node with data value 4, pointed by its local variable `new`, having reached line 3. Thread $\mathtt{th}_3$ has just called the `push` method. Thread $\mathtt{th}_2$ has reached line 12 in the execution of the `pop` method, and has just assigned `youngest` to the first node in the list pointed to by `pools`[3] which is not logically removed (in this case it is the last node of that list). The observer has two registers $\mathtt{x}_1$ and $\mathtt{x}_2$, which are assigned the values 4 and 2, respectively.

We verify the algorithm using a symbolic representation that is analogous to the one used for skiplists. There are two main differences.

– Since the array `pools` is global, all threads can reach all lists in the heap (the only cells that cannot be reached by all threads are new cells that are not yet inserted).
– We therefore represent the view of a thread by a thread-dependent abstraction of thread indices, which index the array `pools`. In the view of a thread, the index of the list where it is currently active is abstracted to `me`, and all other indices are abstracted to `ot`. The currently active index is taken to be the thread index for a thread performing a `push`, the value of `i` for a thread executing in the **for** loop of `pop`, and the value of `k` after that loop.

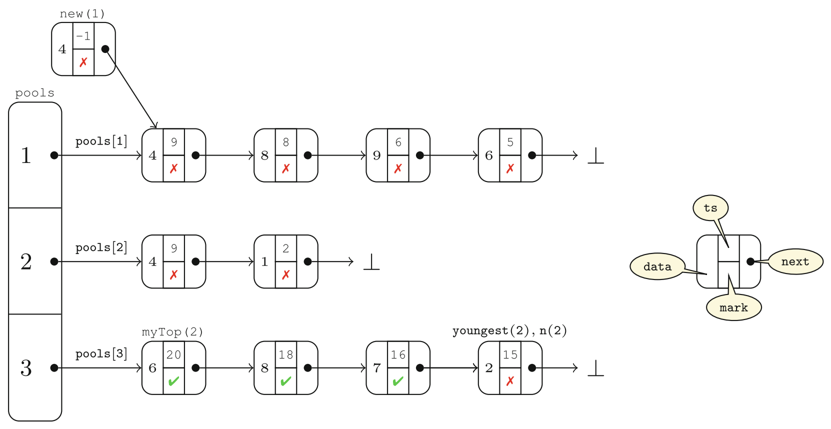


Fig. 9. A possible heap state of TS stack with three threads.

In the definition of tags, the only global variables that can occur in the fields **reachfrom** and **reachto** are therefore **pools**[**me**] and **pools**[**other**]. The data abstraction represents (i) for each cell, the set of observer registers, whose values are equal to the **datafield**, (ii) for each timestamp and observer register $\mathtt{x}_i$, the possible orderings between this timestamp and the timestamp of an $\mathtt{x}_i$-cell.

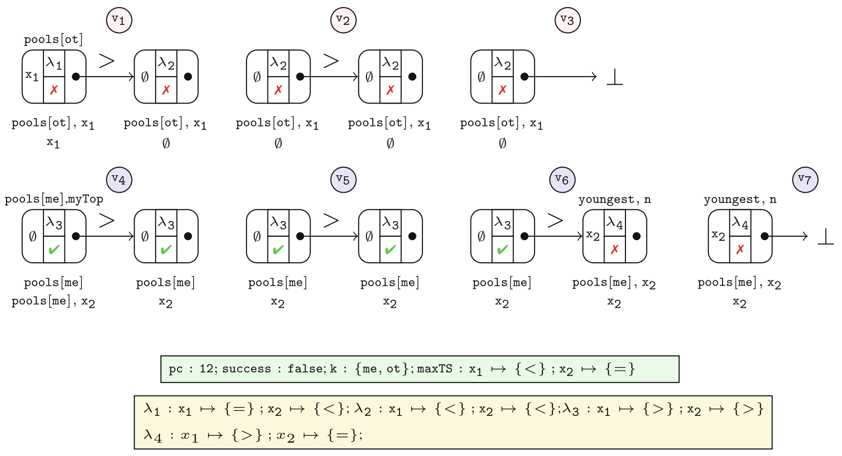

v1
pools[ot]
pools[ot], x1
x1
pools[ot], x1
∅
v2
v3
v4
pools[me],myTop
pools[me]
pools[me], x2
v5
v6
v7
youngest, n
pc : 12; success : false; k : {me, ot}; maxTS : x1 ↦ {<} ; x2 ↦ {=}
λ1 : x1 ↦ {=} ; x2 ↦ {<}; λ2 : x1 ↦ {<} ; x2 ↦ {<};λ3 : x1 ↦ {>} ; x2 ↦ {>}
λ4 : x1 ↦ {>} ; x2 ↦ {=};

Fig. 10. Fragment abstraction

Figure 10 shows a set of fragments that is satisfied wrp. to $\mathtt{th}_2$ by the configuration in Fig. 9. There are 7 fragments, named $\mathtt{v}_1, \ldots, \mathtt{v}_7$. Consider the tag which occurs in fragment $\mathtt{v}_7$. This tag is an abstraction of the bottom-rightmost heap cell in Fig. 9, The different non-pointer fields are represented as follows.

– The `data` field of the tag (to the left) abstracts the data value 2 to the set of observer registers with that value: in this case $\mathtt{x}_2$.
– The `ts` field (at the top) abstracts the timer value 15 to the possible relations with `ts`-fields of heap cells with the same data value as each observer registers. Recall that observer registers $\mathtt{x}_1$ and $\mathtt{x}_2$ have values 4 and 2, respectively. There are three heap cells with `data` field value 4, all with a `ts` value less than 15. There is one heap cell with `data` field value 2, having `ts` value 15. Consequently, the abstraction of the `ts` field maps $\mathtt{x}_1$ to $\{>\}$ and $\mathtt{x}_2$ to $\{=\}$: this is the mapping λ_4 in Fig. 10.
– The `mark` field assumes values from a small finite domain and is represented precisely as in concrete heap cells.

Symbolic Postcondition Computation. The symbolic postcondition computation is similar to that for skiplists. Main differences are as follows.

– Whenever a thread performing `pop` moves from one iteration of the `for` loop to the next, the abstraction must consider to swap between the abstractions `me` and `ot`.
– In interference steps, we must consider that the abstraction `me` for the interfering thread may have to be changed into `ot`. Furthermore, the abstractions `me` for two `push` methods cannot coincide, since each thread pushes only to its own list.

6 Experimental Results

Based on our framework, we have implemented a tool in OCaml, and used it for verifying various kinds of concurrent data structures implementation of stacks, priority queues, queues and sets. All of them are based on heap structures. There are three types of heap structures we consider in our experiments.

Algorithms	Time (s)		Algorithms	Time (s)	
	a	b		a	b
Treiber stack [35]	18	0.18	O′Hearn set [28]	88	12
MS lock-free queue [27]	22	21	HM lock-free set [19]	120	462
DGLM queue [13]	16	16	Harris lock-free set [15]	950	1512
Vechev-CAS set [40]	86	24	Unordered set [43]	1230	2301
Vechev-DCAS set [40]	16	16	TS stack [11]	176	
Michael lock-free set [25]	178	110	TS queue [11]	101	
Pessimistic set [19]	30	1.51	Lock-free skiplist [19]	1992	
Optimistic set [19]	25	60	Lock-based skiplist [18]	500	
Lazy set [17]	34	289	Priority queue skiplist 1 [23]	1320	
			Priority queue skiplist 2 [22]	599	

Fig. 11. Times for verifying concurrent data structure implementations. Column **a** shows the verification times for our tool based on fragment abstraction. Column **b** shows the verification times for the tool for SLLs in our previous work [3]

Singly-linked list benchmarks: These benchmarks include stacks, queues and sets algorithms which are the well-known in the literature. The challenge is that in some set implementation, the linearization points are not fixed, they depended on the future of each execution. The sets with non fixed linearization points are the lazy set [20], lock-free sets of *HM* [22], *Harris* [17], *Michael* [29], and unordered set of [48]. By using observers and controllers in our previous work [3]. Our approach is simple and strong enough to verify these singly-linked list benchmarks.

Skiplist benchmarks: We consider four skiplist algorithms including the lock-based skiplist set [31], the lock-free skiplist set which is described in Sect. 2 [22], and two skiplist-based priority queues [26,27]. One challenge for verifying these algorithms is to deal with unbounded number of levels. In addition, in the lock-free skiplist [22] and priority queue [26], the skiplist shape is not well formed, meaning that each higher level list need not be a sub-list of lower level lists. These algorithms have not been automatically verified in previous work. By applying our fragment abstraction, to the best of our knowledge, we provide first framework which can automatically verify these concurrent skiplists algorithms.

Arrays of singly-linked list benchmarks: We consider two challenging timestamp algorithms in [12]. There are two challenges when verifying these algorithm. The first challenge is how to deal with an unbounded number of SLLs, and the second challenge is that the linearization points of the algorithms are not fixed, but depend on the future of each execution. By combining our fragment abstraction with the observers for stacks and queues in [7], we are able to verify these two algorithms automatically. The observers are crucial for achieving automation, since they enforce the weakest possible ordering constraints that are necessary for proving linearizability, thereby making it possible to use a less precise abstraction.

Running Times. The experiments were performed on a desktop 2.8 GHz processor with 8 GB memory. The results are presented in Fig. 11, where running times are given in seconds. Column a shows the verification times of our tool, whereas column b shows the verification times for algorithms based on SLLs, using the technique in our previous work [3]. In our experiments, we run the tool together with an observer in [1,7] and controllers in [3] to verify linearizability of the algorithms. All experiments start from the initial heap, and end either when the analysis reaches a fixed point or when a violation of safety properties or linearizability is detected. As can be seen from the table, the verification times vary in the different examples. This is due to the types of shapes that are produced during the analysis. For instance, skiplist algorithms have much longer verification times. This is due to the number of pointer variables and their complicated shapes. In contrast, other algorithms produce simple shape patterns and hence they have shorter verification times.

Error Detection. In addition to establishing correctness of the original versions of the benchmark algorithms, we tested our tool with intentionally inserted bugs.

For example, we omitted setting time statement in line 5 of the `push` method in the TS stack algorithm, or we omitted the `CAS` statements in lock-free algorithms. The tool, as expected, successfully detected and reported the bugs.

7 Conclusions

We have presented a novel shape abstraction, called fragment abstraction, for automatic verification of concurrent data structure implementations that operate on different forms of dynamically allocated heap structures, including singly-linked lists, skiplists, and arrays of singly-linked lists. Our approach is the first framework that can automatically verify concurrent data structure implementations that employ skiplists and arrays of singly linked lists, at the same time as handling an unbounded number of concurrent threads, an unbounded domain of data values (including timestamps), and an unbounded shared heap. We showed fragment abstraction allows to combine local and global reachability information to allow verification of the functional behavior of a collection of threads.

As future work, we intend to investigate whether fragment abstraction can be applied also to other heap structures, such as concurrent binary search trees.

References

1. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 324–338. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_23
2. Abdulla, P.A., Holík, L., Jonsson, B., Trinh, C.Q., et al.: Verification of heap manipulating programs with ordered data by extended forest automata. Acta Inf. **53**(4), 357–385 (2016)
3. Abdulla, P.A., Jonsson, B., Trinh, C.Q.: Automated verification of linearization policies. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 61–83. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_4
4. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_49
5. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 399–413. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_37
6. Bingham, J., Rakamarić, Z.: A logic and decision procedure for predicate abstraction of heap-manipulating programs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 207–221. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_14
7. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: On reducing linearizability to state reachability. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015, Part II. LNCS, vol. 9135, pp. 95–107. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_8

8. Bouajjani, A., Emmi, M., Enea, C., Mutluergil, S.O.: Proving linearizability using forward simulations. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017, Part II. LNCS, vol. 10427, pp. 542–563. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_28
9. Chakraborty, S., Henzinger, T.A., Sezgin, A., Vafeiadis, V.: Aspect-oriented linearizability proofs. Log. Methods Comput. Sci. **11**(1) (2015)
10. Chang, B.-Y.E., Rival, X., Necula, G.C.: Shape analysis with structural invariant checkers. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 384–401. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_24
11. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set algorithm. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 475–488. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_44
12. Dodds, M., Haas, A., Kirsch, C.: A scalable, correct time-stamped stack. In: POPL, pp. 233–246. ACM (2015)
13. Doherty, S., Detlefs, D., Groves, L., Flood, C., et al.: DCAS is not a silver bullet for nonblocking algorithm design. In: SPAA 2004, pp. 216–224. ACM (2004)
14. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30232-2_7
15. Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 215–237. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_13
16. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: PODC 2004, pp. 50–59. ACM (2004)
17. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Welch, J. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45414-4_21
18. Haziza, F., Holík, L., Meyer, R., Wolff, S.: Pointer race freedom. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 393–412. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_19
19. Heinen, J., Noll, T., Rieger, S.: Juggrnaut: graph grammar abstraction for unbounded heap structures. ENTCS **266**, 93–107 (2010)
20. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 3–16. Springer, Heidelberg (2006). https://doi.org/10.1007/11795490_3
21. Herlihy, M., Lev, Y., Luchangco, V., Shavit, N.: A simple optimistic skiplist algorithm. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 124–138. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72951-8_11
22. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, San Francisco (2008)
23. Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., Vojnar, T.: Fully automated shape analysis based on forest automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 740–755. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_52

24. Khyzha, A., Dodds, M., Gotsman, A., Parkinson, M.: Proving linearizability using partial orders. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 639–667. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_24
25. Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: PLDI, pp. 459–470. ACM (2013)
26. Lindén, J., Jonsson, B.: A skiplist-based concurrent priority queue with minimal memory contention. In: Baldoni, R., Nisse, N., van Steen, M. (eds.) OPODIS 2013. LNCS, vol. 8304, pp. 206–220. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03850-6_15
27. Lotan, I., Shavit, N.: Skiplist-based concurrent priority queues. In: IPDPS, pp. 263–268. IEEE (2000)
28. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, M.: Predicate abstraction and canonical abstraction for singly-linked lists. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 181–198. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_13
29. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA, pp. 73–82 (2002)
30. Michael, M., Scott, M.: Correction of a memory management method for lock-free data structures. Technical report TR599, University of Rochester, Rochester, NY, USA (1995)
31. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC, pp. 267–275. ACM (1996)
32. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: PODC, pp. 85–94 (2010)
33. Sagiv, S., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002)
34. Sánchez, A., Sánchez, C.: Formal verification of skiplists with arbitrary many levels. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 314–329. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_23
35. Schellhorn, G., Derrick, J., Wehrheim, H.: A sound and complete proof technique for linearizability of concurrent data structures. ACM Trans. Comput. Log. **15**(4), 31:1–37 (2014)
36. Segalov, M., Lev-Ami, T., Manevich, R., Ganesan, R., Sagiv, M.: Abstract transformers for thread correlation analysis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 30–46. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10672-9_5
37. Singh, V., Neamtiu, I., Gupta, R.: Proving concurrent data structures linearizable. In: ISSRE, pp. 230–240. IEEE (2016)
38. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. J. Parallel Distrib. Comput. **65**(5), 609–627 (2005)
39. Treiber, R.: Systems programming: Coping with parallelism. Technical report RJ5118, IBM Almaden Res. Ctr. (1986)
40. Turon, A.J., Thamsborg, J., Ahmed, A., Birkedal, L., Dreyer, D.: Logical relations for fine-grained concurrency. In: POPL 2013, pp. 343–356. ACM (2013)
41. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge (2008)
42. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_40

43. Černý, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 465–479. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_41
44. Vechev, M.T., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: PLDI, pp. 125–135. ACM (2008)
45. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Păsăreanu, C.S. (ed.) SPIN 2009. LNCS, vol. 5578, pp. 261–278. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02652-2_21
46. Wachter, B., Westphal, B.: The spotlight principle. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 182–198. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69738-1_13
47. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_36
48. Zhang, K., Zhao, Y., Yang, Y., Liu, Y., Spear, M.: Practical non-blocking unordered lists. In: Afek, Y. (ed.) DISC 2013. LNCS, vol. 8205, pp. 239–253. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41527-2_17

Security

Reasoning About a Machine with Local Capabilities
Provably Safe Stack and Return Pointer Management

Lau Skorstengaard[1(✉)], Dominique Devriese[2], and Lars Birkedal[1]

[1] Aarhus University, Aarhus, Denmark
{lau,birkedal}@cs.au.dk
[2] imec-DistriNet, KU Leuven, Leuven, Belgium
dominique.devriese@cs.kuleuven.be

Abstract. Capability machines provide security guarantees at machine level which makes them an interesting target for secure compilation schemes that provably enforce properties such as control-flow correctness and encapsulation of local state. We provide a formalization of a representative capability machine with local capabilities and study a novel calling convention. We provide a logical relation that semantically captures the guarantees provided by the hardware (a form of capability safety) and use it to prove control-flow correctness and encapsulation of local state. The logical relation is not specific to our calling convention and can be used to reason about arbitrary programs.

1 Introduction

Compromising software security is often based on attacks that break programming language properties relied upon by software authors, such as control-flow correctness, local-state encapsulation, etc. Commodity processors offer little support for defending against such attacks: they offer security primitives with only coarse-grained memory protection and limited compartmentalization scalability. As a result, defenses against attacks on control-flow correctness and local-state encapsulation are either limited to only certain common forms of attacks (leading to an attack-defense arms race) and/or rely on techniques like machine code rewriting [1,2], machine code verification [3], virtual machines with a native stack [4] or randomization [5]. The latter techniques essentially emulate protection techniques on existing hardware, at the cost of performance, system complexity and/or security.

Capability machines are a type of processors that remediate these limitations with a better security model at the hardware level. They are based on old ideas [6–8], but have recently received renewed interest; in particular, the CHERI project has proposed new ideas and ways of tackling practical challenges like backwards compatibility and realistic OS support [9,10]. Capability machines tag every word (in the register file and in memory) to enforce a strict separation between numbers and capabilities (a kind of pointers that carry authority). Memory capabilities

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 475–501, 2018.
https://doi.org/10.1007/978-3-319-89884-1_17

carry the authority to read and/or write to a range of memory locations. There is also a form of *object capabilities*, which represent the authority to invoke a piece of code without exposing the code's encapsulated private state (e.g., the M-Machine's enter capabilities or CHERI's sealed code/data pairs).

Unlike commodity processors, capability machines lend themselves well to enforcing local-state encapsulation. Potentially, they will enable compilation schemes that enforce this property in an efficient but also 100% watertight way (ideally evidenced by a mathematical proof, guaranteeing that we do not end up in a new attack-defense arms race). However, a lot needs to happen before we get there. For example, it is far from trivial to devise a compilation scheme adapted to the details of a specific source language's notion of encapsulation (e.g., private member variables in OO languages often behave quite differently than private state in ML-like languages). And even if such a scheme were defined, a formal proof depends on a formalization of the encapsulation provided by the capability machine at hand.

A similar problem is the enforcement of control-flow correctness on capability machines. An interesting approach is taken in CheriBSD [9]: the standard contiguous C stack is split into a central, trusted stack, managed by trusted call and return instructions, and disjoint, private, per-compartment stacks. To prevent illegal use of stack references, the approach relies on *local capabilities*, a type of capabilities offered by CHERI to *temporarily* relinquish authority, namely for the duration of a function invocation whereafter the capability can be revoked. However, details are scarce (how does it work precisely? what features are supported?) and a lot remains to be investigated (e.g., combining disjoint stacks with cross-domain function pointers seems like it will scale poorly to large numbers of components?). Finally, there is no argument that the approach is watertight and it is not even clear what security property is targeted exactly.

In this paper, we make two main contributions: (1) an alternative calling convention that uses local capabilities to enforce stack frame encapsulation and well-bracketed control flow, and (2) perhaps more importantly, we adapt and apply the well-studied techniques of step-indexed Kripke logical relations for reasoning about code on a representative capability machine with local capabilities in general and correctness and security of the calling convention in particular. More specifically, we make the following contributions:

- We formalize a simple but representative capability machine featuring local capabilities and its operational semantics (Sect. 2).
- We define a novel calling convention enforcing control-flow correctness and encapsulation of stack frames (Sect. 3). It relies solely on local capabilities and does not require OS support (like a trusted stack or call/return instructions). It supports higher-order cross-component calls (e.g., cross-component function pointers) and can be efficient assuming only one additional piece of processor support: an efficient instruction for clearing a range of memory.
- We present a novel step-indexed Kripke logical relation for reasoning about programs on the capability machine. It is an untyped logical relation, inspired by previous work on object capabilities [11]. We prove an analogue of the

standard fundamental theorem of logical relations—to the best of our knowledge, our theorem is the most general and powerful formulation of the formal guarantees offered by a capability machine (a form of capability safety [11,12]), including the specific guarantees offered for local capabilities. It is very general and not tied to our calling convention or a specific way of using the system's capabilities. We are the first to apply these techniques for reasoning about capability machines and we believe they will prove useful for many other purposes than our calling convention.
– We introduce two novel technical ideas in the unary, step-indexed Kripke logical relation used to formulate the above theorem: the use of a *single* orthogonal closure (rather than the earlier used biorthogonal closure) and a variant of Dreyer et al. [13]'s public and private future worlds [13] to express the special nature of local capabilities. The logical relation and the fundamental theorem expressing capability safety are presented in Sect. 4.
– We demonstrate our results by applying them to challenging examples, specifically constructed to demonstrate local-state encapsulation and control-flow correctness guarantees in the presence of cross-component function pointers (Sect. 5). The examples demonstrate both the power of our formulation of capability safety and our calling convention.

For reasons of space, some details and all proofs have been omitted; please refer to the technical appendix [14] for those.

2 A Capability Machine with Local Capabilities

In this paper, we work with a formal capability machine with all the characteristics of real capability machines, as well as local capabilities much like CHERI's. Otherwise, it is kept as simple as possible. It is inspired by both the M-Machine [6] and CHERI [9]. To avoid uninteresting details, we assume an infinite address space and unbounded integers.

We define the syntax of our capability machine in Fig. 1. We assume an infinite set of addresses Addr and define machine words as either integers or capabilities of the form $((perm, g), base, end, a)$. Such a capability represents the authority to execute permissions $perm$ on the memory range $[base, end]$, together with a current address a and a locality tag g indicating whether the capability is global or local. There is no notion of pointers other than capabilities, so we will use the terms interchangeably. The available permissions and their ordering are depicted in Fig. 3: the permissions include null permission (O), readonly (RO), read/write (RW), read/execute (RX) and read-/write/execute (RWX) permissions. Additionally, there are three special permissions: read/write-local (RWL), read/write-local/execute (RWLX) and enter (E), which we will explain below.

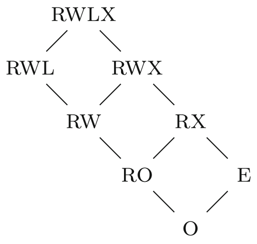


Fig. 3. Permission hierarchy

$$
\begin{array}{rrcl@{\qquad}rrcl}
a \in & \mathrm{Addr} & \overset{\text{def}}{=} & \mathbb{N} & r \in & \mathrm{RegName} & ::= & \mathrm{pc} \mid r_0 \mid r_1 \mid \ldots \\
w \in & \mathrm{Word} & \overset{\text{def}}{=} & \mathbb{Z} + \mathrm{Cap} & \mathit{reg} \in & \mathrm{Reg} & \overset{\text{def}}{=} & \mathrm{RegName} \to \mathrm{Word} \\
\mathit{perm} \in & \mathrm{Perm} & ::= & \textsc{o} \mid \textsc{ro} \mid \textsc{rw} \mid \textsc{rwl} \mid & m \in & \mathrm{Mem} & \overset{\text{def}}{=} & \mathrm{Addr} \to \mathrm{Word} \\
 & & & \textsc{rx} \mid \textsc{e} \mid \textsc{rwx} \mid \textsc{rwlx} & \Phi \in & \mathrm{ExecConf} & \overset{\text{def}}{=} & \mathrm{Reg} \times \mathrm{Mem} \\
g \in & \mathrm{Global} & ::= & \mathrm{global} \mid \mathrm{local} & \mathit{ms} \in & \mathrm{MemSeg} & ::= & \mathrm{Addr} \rightharpoonup \mathrm{Word}
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{Conf} & ::= & \mathrm{ExecConf} + \{\mathit{failed}\} + \{\mathit{halted}\} \times \mathrm{Mem} \\
\mathrm{Cap} & ::= & \{((\mathit{perm}, g), b, e, a) \mid b, a \in \mathrm{Addr}, e \in \mathrm{Addr} \cup \{\infty\}\}
\end{array}
$$

$$
\begin{array}{rcl}
r & \in & \mathbb{Z} + \mathrm{RegName} \\
i & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ r \mid \texttt{move}\ r\ r \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ r \mid \texttt{plus}\ r\ r\ r \mid \texttt{minus}\ r\ r\ r \mid \\
 & & \texttt{lt}\ r\ r\ r \mid \texttt{lea}\ r\ r \mid \texttt{restrict}\ r\ r \mid \texttt{subseg}\ r\ r\ r \mid \texttt{isptr}\ r\ r \mid \texttt{getl}\ r\ r \mid \\
 & & \texttt{getp}\ r\ r \mid \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \texttt{fail} \mid \texttt{halt}
\end{array}
$$

Fig. 1. The syntax of our capability machine assembly language.

$$
\Phi \to \begin{cases} [\![\mathit{decode}(n)]\!]\,(\Phi) & \text{if } \Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), b, e, a) \text{ and } b \leq a \leq e \\ & \quad \text{and } \mathit{perm} \in \{\textsc{rx}, \textsc{rwx}, \textsc{rwlx}\} \text{ and } \Phi.\mathrm{mem}(a) = n \\ \mathit{failed} & \text{otherwise} \end{cases}
$$

$$
\mathit{updPc}(\Phi) = \begin{cases} \Phi[\mathrm{reg.pc} \mapsto \mathit{newPc}] & \text{if } \Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), b, e, a) \\ & \quad \text{and } \mathit{newPc} = ((\mathit{perm}, g), b, e, a+1) \\ \mathit{failed} & \text{otherwise} \end{cases}
$$

i	$[\![i]\!]\,(\Phi)$	Conditions
`fail`	*failed*	
`halt`	$(\mathit{halted}, \Phi.\mathrm{mem})$	
`move` $r_1\ r_2$	$\mathit{updPc}(\Phi[\mathrm{reg}.r_1 \mapsto w])$	$r_2 \in \mathrm{Reg} \Rightarrow w = \Phi.\mathrm{reg}(r_2)$ and $r_2 \in \mathbb{Z} \Rightarrow w = r_2$
`load` $r_1\ r_2$	$\mathit{updPc}(\Phi[\mathrm{reg}.r_1 \mapsto w])$	$\Phi.\mathrm{reg}(r_2) = ((\mathit{perm}, g), b, e, a)$ and $w = \Phi.\mathrm{mem}(a)$ and $b \leq a \leq e$ and $\mathit{perm} \in \{\textsc{rwx}, \textsc{rwlx}, \textsc{rx}, \textsc{rw}, \textsc{rwl}, \textsc{ro}\}$
`restrict` $r_1\ r_2$	$\mathit{updPc}(\Phi[\mathrm{reg}.r_1 \mapsto w])$	$\Phi.\mathrm{reg}(r_2) = ((\mathit{perm}, g), b, e, a)$ and $(\mathit{perm}', g') = \mathit{decodePermPair}(\Phi.\mathrm{reg}(r_2))$ and $(\mathit{perm}', g') \sqsubseteq (\mathit{perm}, g)$ and $w = ((\mathit{perm}', g'), b, e, a)$
`geta` $r_1\ r_2$	$\mathit{updPc}(\Phi[\mathrm{reg}.r_1 \mapsto a])$	$\Phi.\mathrm{reg}(r_2) = ((\_, \_), \_, \_, a)$
`jmp` r	$\Phi[\mathrm{reg.pc} \mapsto \mathit{newPc}]$	if $\Phi.\mathrm{reg}(r) = ((\textsc{e}, g), b, e, a)$, then $\mathit{newPc} = ((\textsc{rx}, g), b, e, a)$ otherwise $\mathit{newPc} = \Phi.\mathrm{reg}(r)$
`store` $r_1\ r_2$	$\mathit{updPc}(\Phi[\mathrm{mem}.a \mapsto w])$	$\Phi.\mathrm{reg}(r_1) = ((\mathit{perm}, g), b, e, a)$ and $\mathit{perm} \in \{\textsc{rwx}, \textsc{rwlx}, \textsc{rw}, \textsc{rwl}\}$ and $b \leq a \leq e$ and $w = \Phi.\mathrm{reg}(r_2)$ and if $w = ((\_, \mathrm{local}), \_, \_, \_)$, then $\mathit{perm} \in \{\textsc{rwlx}, \textsc{rwl}\}$
…		
_	*failed*	otherwise

Fig. 2. An excerpt from the operational semantics.

We assume a finite set of register names RegName. We define register files *reg* and memories *ms* as functions mapping register names resp. addresses to words. The state of the entire machine is represented as a configuration that is either a running state $\Phi \in$ ExecConf containing a memory and a register file, or a failed or halted state, where the latter keeps hold of the final state of memory.

The machine's instruction set is rather basic. Instructions i include relatively standard jump (`jmp`), conditional jump (`jnz`) and move (`move`, copies words between registers) instructions. Also familiar are load and store instructions for reading from and writing to memory (`load` and `store`) and arithmetic addition operators (`lt` (less than), `plus` and `minus`, operating only on numbers). There are three instructions for modifying capabilities: `lea` (modifies the current address), `restrict` (modifies the permission and local/global tag) and `subseg` (modifies the range of a capability). Importantly, these instructions take care that the resulting capability always carries less authority than the original (e.g. `restrict` will only weaken a permission). Finally, the instruction `isptr` tests whether a word is a capability or a number and instructions `getp`, `getl`, `getb`, `gete` and `geta` provide access to a capability's permissions, local/global tag, base, end and current address, respectively.

Figure 2 shows an excerpt of the operational semantics for a few representative instructions. Essentially, a configuration Φ either decodes and executes the instruction at Φ.reg(pc) if it is executable and its address is in the valid range or otherwise fails. The table in the figure shows for instructions i the result of executing them in configuration Φ. `fail` and `halt` obviously fail and halt respectively. `move` simply modifies the register file as requested and updates the pc to the next instruction using the meta-function *updPc*.

The `load` instruction loads the contents of the requested memory location into a register, but only if the capability has appropriate authority (i.e. read permission and an appropriate range). `restrict` updates a capability's permissions and global/local tag in the register file, but only if the new permissions are weaker than the original. It also never turns local capabilities into global ones. `geta` queries the current address of a capability and stores it in a register.

The `jmp` instruction updates the program counter to a requested location, but it is complicated by the presence of *enter capabilities*, modeled after the M-Machine's [6]. Enter capabilities cannot be used to read, write or execute and their address and range cannot be modified. They can only be used to jump to, but when that happens, their permission changes to RX. They can be used to represent a kind of closures: an opaque package containing a piece of code together with local encapsulated state. Such a package can be built as an enter capability $c = ((\text{E}, g), b, e, a)$ where the range $[b, a - 1]$ contains local state (data or capabilities) and $[a, e]$ contains instructions. The package is opaque to an adversary holding c but when c is jumped to, the instructions can start executing and have access to the local data through the updated version of c that is then in pc.

Finally, the `store` instruction updates the memory to the requested value if the capability has write authority for the requested location. However, the instruction is complicated by the presence of *local capabilities*, modeled after the ones in the CHERI processor [9]. Basically, local capabilities are special in that they can only be kept in registers, i.e. they cannot be stored to memory. This means that local capabilities can be *temporarily* given to an adversary, for the duration of an invocation: if we take care to clear the capability from the register file after control is passed back to us, they will not have been able to store the capability. However, there is one exception to the rule above: local capabilities can be stored to memory for which we have a capability with write-local authority (i.e. permission RWL or RWLX). This is intended to accommodate a stack, where register contents can be stored, including local capabilities. As long as all capabilities with write-local authority are themselves local and the stack is cleared after control is passed back by the adversary, we will see that this does not break the intended behavior of local capabilities.

We point out that our local capabilities capture only a part of the semantics of local capabilities in CHERI. Specifically, in addition to the above, CHERI's default implementation of the CCall exception handler forbids local capabilities from being passed across module boundaries. Such a restriction fundamentally breaks our calling convention, since we pass around local return pointers and stack capabilities. However, CHERI's CCall is not implemented in hardware, but in software, precisely to allow experimenting with alternative models like ours.

In order to have a reasonably realistic system, we use a simple model of linking where a program has access to a linking table that contains capabilities for other programs. We also assume malloc to be part of the trusted computing base satisfying a certain specification. Malloc and linking tables are described further in the next section, but we refer to the technical appendix [14] for full details.

3 Stack and Return Pointer Management Using Local Capabilities

One of the contributions in this paper is a demonstration that local capabilities on a capability machine support a calling convention that enforces control-flow correctness in a way that is provably watertight, potentially efficient, does not rely on a trusted central stack manager and supports higher-order interfaces to an adversary, where an adversary is just some unknown piece of code. In this section, we explain this convention's high-level approach, the security measures to be taken in a number of situations (motivating each separately with a summary table at the end). After that, we define a number of reusable macro-instructions that can be used to conveniently apply the proposed convention in subsequent examples.

The basic idea of our approach is simple: we stick to a single, rather standard, C stack and register-passed stack and return pointers, much like a standard C calling convention. However, to prevent various ways of misusing this basic scheme, we put local capabilities to work and take a number of not-always-obvious safety measures. The safety measures are presented in terms of what *we* need to do to protect ourselves against an *adversary*, but this is only for presentation purposes as our code assumes no special status on the machine. In fact, an adversary can apply the same safety measures to protect themselves against us. In the next paragraphs, we will explain the issues to be considered in all the relevant situations: when (1) starting our program, (2) returning to the adversary, (3) invoking the adversary, (4) returning from the adversary, (5) invoking an adversary callback and (6) having a callback invoked by the adversary.

Program Start-Up. We assume that the language runtime initializes the memory as follows: a contiguous array of memory is reserved for the stack, for which we receive a stack pointer in a special register r_{stk}. We stress that the stack is not built-in, but merely an abstraction we put on this piece of the memory. The stack pointer is local and has RWLX permission. Note that this means that we will be placing and executing instructions on the stack. Crucially, the stack is the only part of memory for which the runtime (including malloc, loading, linking) will ever provide RWLX or RWL capabilities. Additionally, our examples typically also assume some memory to store instructions or static data. Another part of memory (called the heap) is initially governed by malloc and at program start-up, no other code has capabilities for this memory. Malloc hands out RWX capabilities for allocated regions as requested (no RWLX or RWL permissions). For simplicity, we assume that memory allocated through malloc cannot be freed.

Returning to the Adversary. Perhaps the simplest situation is returning to the adversary after they invoked our code. In this case, we have received a return pointer from them, and we just need to jump to it as usual. An obvious security measure to take care of is properly clearing the non-return-value registers before we jump (since they may contain data or capabilities that the adversary should not get access to). Additionally, we may have used the stack for various purposes (register spilling, storing local state when invoking other functions etc.), so we also need to clear that data before returning to the adversary.

However, if we are returning from a function that has itself invoked adversary code, then clearing the used part of the stack is not enough. The *unused* part of the stack may also contain data and capabilities, left there by the adversary, including local capabilities since the stack is write-local. As we will see later, we rely on the fact that the adversary cannot keep hold of local capabilities when they pass control to the trusted code and receive control back. In this case, the adversary could use the unused part of the stack to store local pointers and load them from there after they get control back. To prevent this, we need to clear (i.e. overwrite with zeros) the entire part of the stack that the adversary has had access to, not just the parts that we have used ourselves. Since we may be talking about a large part of memory, this requirement is the most problematic aspect of our calling convention for performance, but see Sect. 6 for how this might be mitigated.

Invoking the Adversary. A slightly more complex case is invoking the adversary. As above, we clear all the non-argument registers, as well as the part of the stack that we are not using (because, as above, it may contain local capabilities from previously executed code that the adversary could exploit in the same way). We leave a copy of the stack pointer in r_{stk}, but only after we have used the `subseg` instruction to shrink its authority to the part that we are not using ourselves.

In one of the registers, we also provide a return pointer, which must be a local capability. If it were global, the adversary would be able to store away the return pointer in a global data structure (i.e. there exists a global capability for it), and jump to it later, in circumstances where this should not be possible. For example, they could store the return pointer, legally jump to it a first time, wait to be invoked again and then jump to the old return pointer a second time, instead of the new return pointer received for the second invocation. Similarly, they could store the return pointer, invoke a function in our code, wait for us to invoke them again and then jump to the old return pointer rather than the new one, received for the second invocation. By making the return pointer local, we prevent such attacks: the adversary can only store local capabilities through write-local capabilities, which means (because of our assumptions above): on the stack. Since the stack pointer itself is also local, it can also only be stored on the stack. Because we clear the part of the stack that the adversary has had access to before we pass control back, there is no way for them to recover either of these local capabilities.

Note that storing stack pointers for use during future invocations would also be dangerous in itself, i.e. not just because it can be used to store return pointers. Imagine the adversary stores their stack pointer, invokes trusted code that uses part of the stack to store private data and then invokes the adversary again with a stack pointer restricted to exclude the part containing the private data. If the adversary had a way of keeping hold of their old stack pointer, it could access the private data stored there by the trusted code and break local-state encapsulation.

Returning from the Adversary. So return pointers must be passed as local capabilities. But what should their permissions be, what memory should they point to and what should that memory (the activation record) contain? Let us answer the last question first by considering what should happen when the adversary jumps to a return pointer. In that case, the program counter should be restored to the instruction after the jump to the adversary, so the activation record should store this old program counter. Additionally, the stack pointer should also be restored to its original value. Since the adversary has a more restricted authority over the stack than the code making the call, we cannot hope to reconstruct the original stack pointer from the stack pointer owned by the adversary. Instead, it should be stored as part of the activation record.

Clearly, neither of these capabilities should be accessible by the adversary. In other words, the return pointer provided to the adversary must be a capability that they can jump to but not read from, i.e. an enter capability. To make this work, we construct the activation record as depicted in Fig. 4. The E return pointer has authority over the entire activation record (containing the previous return and stack pointer), and its current address points to a number of restore instructions in the record, so that upon invocation, these instructions are executed and can load the old stack pointer and program counter back into the register file. As the return pointer is an enter pointer, the adversary cannot get

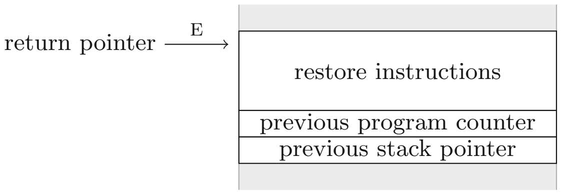


Fig. 4. Structure of an activation record

hold of the activation record's contents, but after invocation, its permission is updated to RX, so the contents become available to the restore instructions.

The final question that remains is: where should we store this activation record? The attentive reader may already see that there is only one possibility: since the activation record contains the old stack pointer, which is local, the activation record can only be constructed in a part of memory where we have write-local access, i.e. on the stack. Note that this means we will be placing and executing instructions on the stack, i.e. it will not just contain code pointers and data. This means that our calling convention should be combined with protection against stack smashing attacks (i.e. buffer overflows on the stack overwriting activation records' contents). Luckily, the capability machine's fine-grained memory protection should make it reasonably easy for a compiler to implement such protection, by making sure that only appropriately bounded versions of the stack pointer are made available to source language code.

Invoking an Adversary Callback. If we have a higher-order interface to the adversary, we may need to invoke an adversary callback. In this case, not so much changes with respect to the situation where we invoke static adversary code. The adversary can provide a callback as a capability for us to jump to, either an E-capability if they want to protect themselves from us or just an RX capability if they are not worried about that. However, there is one scenario that we need to prevent: if they construct the callback capability to point into the stack, it may contain local capabilities that they should not have access to upon invocation of the callback. As before, this includes return and stack pointers from previous stack frames that they may be trying to illegally use inside the callback.

To prevent this, we only accept callbacks from the adversary in the form of global capabilities, which we dynamically check before invoking them (and we fail otherwise). This should not be an overly strict requirement: our own callbacks do not contain local data themselves, so there should be no need for the adversary to construct callbacks on the stack.[1]

Having a Callback Invoked by the Adversary. The above leaves us with perhaps the hardest scenario: how to provide a callback to the adversary. The

[1] Note that it does prevent a legitimate but non-essential scenario where the adversary wants to give us temporary access to a callback not allocated on the stack.

basic idea is that we allocate a block of memory using malloc that we fill with the capabilities and data that the callback needs, as well as some prelude instructions that load the data into registers and jumps to the right code. Note that this implies that no local capabilities can be stored as part of a closure. We can then provide the adversary with an enter-capability covering the allocated block and pointing to the contained prelude instructions. However, the question that remains in this setup is: from where do we get a stack pointer when the callback is invoked?

Our answer is that the adversary should provide it to us, just as we provide them with a stack pointer when we invoke their code. However, it is important that we do not just accept any capability as a stack pointer but check that it is safe to use. Specifically, we check that it is indeed an RWLX capability. Without this check, an adversary could potentially get control over our local stack frame during a subsequent callback by passing us a local RWX capability to a global data structure instead of a proper stack pointer and a global callback for our callback to invoke. If our local state contains no local capabilities, then, otherwise following our calling convention, the callback would not fail and the adversary could use a stored capability for the global data structure to access our local state. To prevent this from happening, we need to make sure the stack capability carries RWLX authority, since the system wide assumption then tells us that the adversary cannot have global capabilities to our local stack.

Calling Convention. With the security measures introduced and motivated, let us summarize our proposed calling convention: *At program start-up* A local RWLX stack pointer resides in register r_{stk}. No global write-local capabilities. *Before returning to the adversary* Clear non-return-value registers. Clear the part of the stack we had access to (not just the part we used). *Before invoking the adversary* Push activation record to the stack. Create return pointer as local E-capability to the instructions in the record. Restrict the stack capability to the unused part and clear it. Clear non-argument registers. *Before invoking an adversary callback* Make sure callback is global. *When invoked by an adversary* Make sure received stack pointer has permission RWLX.

Reusable Macro Instructions. We define a number of reusable macros capturing the calling convention and other conveniences. All macros that use the stack assume a stack pointer in register r_{stk}. The macro `fetch` r *name* fetches the capability related to *name* from the linking table and stores it in register r. The macros `push` r and `pop` r add and remove elements from the stack. The macro `prepstk` r is used when a callback is invoked by the adversary and prepares the received stack pointer by checking that it has permission RWLX. The macro `scall` $r(\overline{r_{args}},\overline{r_{priv}})$ jumps to the capability in register r in the manner described above. That is, it pushes local state (the contents of registers $\overline{r_{priv}}$) and the activation record (return code, return pointer, stack pointer) to the stack, creates an E return pointer, restricts the stack pointer, clears the unused part of the stack, clears the necessary registers and jumps to r. Upon return, the private state is restored. The macro `mclear` r clears all the memory the capability in register r has authority over. The macro `rclear` *regSet* clears all the registers in *regSet*. The macro `reqglob` r checks whether the word in register r is a global capability. The macro `crtcls` $\overline{(x_i, r_i)}$ r allocates a closure where r points to the closure's code and a new environment is allocated (using malloc) where the contents of $\overline{r_i}$ is stored. In the code referred to by r, an implicit fetch happens when an instruction refers to x_i.

The technical appendix [14] contains detailed descriptions of all the macros.

4 Logical Relation

In this section, we formalize the guarantees provided by the capability machine, including the specific guarantees for local capabilities, by means of a step-indexed Kripke logical relation with recursively defined worlds. We use the logical relation in the following section to show local-state encapsulation and control-flow integrity properties for challenging example programs.

4.1 Worlds

A world is a finite map from region names, modeled as natural numbers, to regions that each correspond to an invariant of part of the memory. We have three types of regions: *permanent*, *temporary*, and *revoked*. Each permanent and temporary region contains a state transition system, with public and private transitions, to describe how the invariants are allowed to change over time. In other words, they are protocols for the region's memory. These are similar to what has been used in logical relations for high-level languages [11,13,15]. Protocols imposed by permanent regions stay in place indefinitely. Any capability, local or global, can depend on these protocols. Protocols imposed by temporary regions can be revoked in private future worlds. Doing this may break the safety of local capabilities but not global ones. This means that local capabilities can safely depend on the protocols imposed by temporary regions, but global capabilities cannot, since a global capability may outlive a temporary region that is revoked. This is illustrated in Fig. 5.

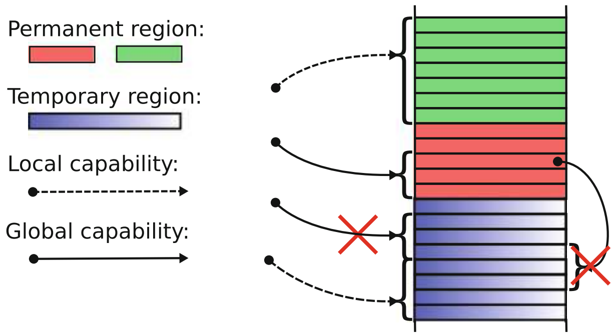


Fig. 5. The relation between local/global capabilities and temporary/permanent regions. The colored fields are regions governing parts of memory. Global capabilities cannot depend on temporary regions.

For technical reasons, we do not actually remove a revoked temporary region from the world, but we turn it into a special revoked region that exists for this purpose. Such a revoked region contains no state transition system and puts no requirements on the memory. It simply serves as a mask for a revoked temporary region. Masking a region like this goes back to earlier work of Ahmed [16] and was also used by Birkedal et al. [17].

Regions are used to define safe memory segments, but this set may itself be world-dependent. In other words, our worlds are defined recursively. Recursive worlds are common in Kripke models and the following lemma uses the method of Birkedal and Bizjak [18]; Birkedal et al. [19] for constructing them. The formulation of the lemma is technical, so we recommend that non-expert readers ignore the technicalities and accept that there exists a set of worlds Wor and two relations $\sqsupseteq^{priv}$ and $\sqsupseteq^{pub}$ satisfying the (recursive) equations in the theorem (where the $\blacktriangleright$ operator can be safely ignored).

Theorem 1. *There exists a c.o.f.e. (complete ordered family of equivalences)* Wor *and preorders* $\sqsupseteq^{priv}$ *and* $\sqsupseteq^{pub}$ *such that* $(\mathrm{Wor}, \sqsupseteq^{priv})$ *and* $(\mathrm{Wor}, \sqsupseteq^{pub})$ *are preordered c.o.f.e.'s, and there exists an isomorphism* ξ *such that*

$$\begin{aligned}
\xi \ &: \mathrm{Wor} \cong \blacktriangleright(\mathbb{N} \xrightarrow{fin} \mathrm{Region})\\
\mathrm{Region} &= \{\mathrm{revoked}\} \uplus\\
&\qquad \{\mathrm{temp}\} \times \mathrm{State} \times \mathrm{Rels} \times (\mathrm{State} \to (\mathrm{Wor} \xrightarrow[\sqsupseteq^{pub}]{mon,\ ne} \mathrm{UPred}(\mathrm{MemSeg}))) \uplus\\
&\qquad \{\mathrm{perm}\} \times \mathrm{State} \times \mathrm{Rels} \times (\mathrm{State} \to (\mathrm{Wor} \xrightarrow[\sqsupseteq^{priv}]{mon,\ ne} \mathrm{UPred}(\mathrm{MemSeg})))
\end{aligned}$$

and for $W, W' \in \mathrm{Wor}$.
$$\begin{aligned}
W' \sqsupseteq^{priv} W &\Leftrightarrow \xi(W') \sqsupseteq^{priv} \xi(W)\\
W' \sqsupseteq^{pub} W &\Leftrightarrow \xi(W') \sqsupseteq^{pub} \xi(W)
\end{aligned}$$

In the above theorem, State $\times$ Rels corresponds to the aforementioned state transition system where Rels contains pairs of relations corresponding to the public and private transitions, and State is an unspecified set that we assume to contain at least the states we use in this paper. The last part of the temporary and permanent regions is a state interpretation function that determines what memory segments the region permits in each state of the state transition system. The different monotonicity requirements in the two interpretation functions reflects how permanent regions rely only on permanent protocols whereas temporary regions can rely on both temporary and permanent protocols. UPred(MemSeg) is the set of step-indexed, downwards closed predicates on memory segments: $\mathrm{UPred}(\mathrm{MemSeg}) = \{A \subseteq \mathbb{N} \times \mathrm{MemSeg} \mid \forall (n, ms) \in A. \forall m \leq n. (m, ms) \in A\}$.

With the recursive domain equation solved, we could take Wor as our notion of worlds, but it is technically more convenient to work with the following definition instead:

$$\mathrm{World} = \mathbb{N} \xrightarrow{fin} \mathrm{Region}$$

Future Worlds. The future world relations model how memory may evolve over time. The *public future world* $W' \sqsupseteq^{pub} W$ requires that $\mathrm{dom}(W') \supseteq \mathrm{dom}(W)$ and $\forall r \in \mathrm{dom}(W).\, W'(r) \sqsupseteq^{pub} W(r)$. That is, in a public future world, new regions may have been allocated, and existing regions may have evolved according to the public future region relation (defined below). The *private future world* relation $W' \sqsupseteq^{priv} W$ is defined similarly, using a private future region relation. The *public future* region relation is the simplest. It satisfies the following properties:

$$\frac{(s, s') \in \phi_{pub}}{(v, s', \phi_{pub}, \phi, H) \sqsupseteq^{pub} (v, s, \phi_{pub}, \phi, H)} \qquad \frac{(\mathrm{temp}, s, \phi_{pub}, \phi, H) \in \mathrm{Region}}{(\mathrm{temp}, s, \phi_{pub}, \phi, H) \sqsupseteq^{pub} \mathrm{revoked}}$$

$$\frac{}{\mathrm{revoked} \sqsupseteq^{pub} \mathrm{revoked}}$$

Both temporary and permanent regions are only allowed to transition according to the public part of their transition system. Additionally, revoked regions must either remain revoked or be replaced by a temporary region. This means that the public future world relations allows us to reinstate a region that has been revoked earlier. The *private future region* relation satisfies:

$$\frac{(s, s') \in \phi}{(v, s', \phi_{pub}, \phi, H) \sqsupseteq^{priv} (v, s, \phi_{pub}, \phi, H)} \qquad \frac{r \in \mathrm{Region}}{r \sqsupseteq^{priv} (\mathrm{temp}, s, \phi_{pub}, \phi, H)}$$

$$\frac{r \in \mathrm{Region}}{r \sqsupseteq^{priv} \mathrm{revoked}}$$

Here, revocation of temporary regions is allowed. In fact, temporary regions can be replaced by an arbitrary other region, not just the special revoked. Conversely, revoked regions may also be replaced by any other region. On the other hand, permanent regions cannot be masked away. They are only allowed to transition according to the private part of the transition system.

Notice that the public future region relation is a subset of the private future region relation.

World Satisfaction. A memory satisfies a world, written $ms :_n W$, if it can be partitioned into disjoint parts such that each part is accepted by an active (permanent or temporary) region. Revoked regions are not taken into account as their memory protocols are no longer in effect.

$$ms :_n W \quad \text{iff} \quad \begin{cases} \exists P : \mathit{active}(W) \to \mathrm{MemSeg}.\, ms = \displaystyle\biguplus_{r \in \mathit{active}(W)} P(r) \ \text{ and} \\ \quad \forall r \in \mathit{active}(W). \\ \qquad \exists H, s.\, W(r) = (\_, s, \_, \_, H) \ \text{ and } \ (n, P(r)) \in H(s)(\xi^{-1}(W)) \end{cases}$$

$$\mathcal{O} : \text{World} \xrightarrow{ne} \text{UPred}(\text{Reg} \times \text{MemSeg})$$

$$\mathcal{O}(W) \stackrel{def}{=} \left\{ (n, (reg, ms)) \;\middle|\; \begin{array}{l} \forall ms_f, mem', i \leq n. \, (reg, ms \uplus ms_f) \rightarrow_i (halted, mem') \Rightarrow \\ \quad \exists W' \sqsupseteq^{priv} W, ms_r, ms'. \\ \qquad mem' = ms' \uplus ms_r \uplus ms_f \text{ and } ms' :_{n-i} W' \end{array} \right\}$$

$$\mathcal{R} : \text{World} \xrightarrow[\sqsupseteq^{pub}]{mon,\, ne} \text{UPred}(\text{Reg})$$

$$\mathcal{R}(W) \stackrel{def}{=} \{(n, reg) \mid \forall r \in \text{RegName} \setminus \{\text{pc}\}. \, (n, reg(r)) \in \mathcal{V}(W)\}$$

$$\mathcal{E} : \text{World} \xrightarrow{ne} \text{UPred}(\text{Word})$$

$$\mathcal{E}(W) \stackrel{def}{=} \left\{ (n, pc) \;\middle|\; \begin{array}{r} \forall n' \leq n, \left(n', reg\right) \in \mathcal{R}(W), ms :_{n'} W. \\ \left(n', (reg[\text{pc} \mapsto pc], ms)\right) \in \mathcal{O}(W) \end{array} \right\}$$

$$\mathcal{V} : \text{World} \xrightarrow[\sqsupseteq^{pub}]{mon,\, ne} \text{UPred}(\text{Word})$$

$$\begin{array}{ll} \mathcal{V}(W) \stackrel{def}{=} & \{(n, i) \mid i \in \mathbb{Z}\} \cup \{(n, ((\textsc{o}, g), b, e, a))\} \cup \\ & \left\{ (n, ((\textsc{rw}, g), b, e, a)) \;\middle|\; \begin{array}{l} (n, (b, e)) \in readCond(g)(W) \text{ and} \\ (n, (b, e)) \in writeCond(\iota^{nwl}, g)(W) \end{array} \right\} \cup \\ & \{(n, ((\textsc{e}, g), b, e, a)) \mid (n, (b, e, a)) \in enterCond(g)(W)\} \cup \\ & \left\{ (n, ((\textsc{rwlx}, g), b, e, a)) \;\middle|\; \begin{array}{l} (n, (b, e)) \in readCond(g)(W) \text{ and} \\ (n, (b, e)) \in writeCond(\iota^{pwl}, g)(W) \text{ and} \\ (n, (\{\textsc{rwlx}, \textsc{rwx}, \textsc{rx}\}, b, e)) \in execCond(g)(W) \end{array} \right\} \\ & \cup \ldots \textit{and so on for permissions } \textsc{ro}, \textsc{rwl}, \textsc{rx}, \textit{and } \textsc{rwx}. \end{array}$$

Fig. 6. The logical relation.

4.2 Logical Relation

The logical relation defines semantically when values, program counters, and configurations are capability safe. The definition is found in Figs. 6 and 7 and we provide some explanations in the following paragraphs. For space reasons, we omit some definitions and explain them only verbally, but precise definitions can be found in the technical appendix [14].

First, the *observation relation* $\mathcal{O}$ defines what configurations we consider safe. A configuration is safe with respect to a world, when the execution of said configuration does not break the memory protocols of the world. Roughly speaking, this means that when the execution of a configuration halts, then there is a private future world that the resulting memory satisfies. Notice that failing is considered safe behavior. In fact, the machine often resorts to failing when an unauthorized access is attempted, such as loading from a capability without read permission. This is similar to Devriese et al. [11]'s logical relation for an untyped language, but unlike typical logical relations for typed languages, which require that programs do not fail.

The *register-file relation* $\mathcal{R}$ defines safe register-files as those that contain safe words (i.e. words in $\mathcal{V}$) in all registers but pc. The *expression relation* $\mathcal{E}$ defines that a word is safe to use as a program counter if it can be plugged into

$$readCond(g)(W) = \left\{ (n, (b, e)) \;\middle|\; \begin{array}{l} \exists r \in localityReg(g, W). \\ \quad \exists [b', e'] \supseteq [b, e].\, W(r) \overset{n}{\subsetapprox} \iota^{pwl}_{b',e'} \end{array} \right\}$$

$$writeCond(\iota, g)(W) = \left\{ (n, (b, e)) \;\middle|\; \begin{array}{l} \exists r \in localityReg(g, W). \\ W(r) \text{ is address-stratified and} \\ \quad \exists [b', e'] \supseteq [b, e].\, W(r) \overset{n-1}{\supsetapprox} \iota_{b',e'} \end{array} \right\}$$

$$execCond(g)(W) = \left\{ (n, (P, b, e)) \;\middle|\; \begin{array}{l} \forall n' < n, W' \sqsupseteq W, a \in [b, e], perm \in P. \\ \quad \big(n', ((perm, g), b, e, a)\big) \in \mathcal{E}(W') \end{array} \right\}$$

$$enterCond(g)(W) = \left\{ (n, (b, e, a)) \;\middle|\; \begin{array}{l} \forall n' < n.\, \forall W' \sqsupseteq W. \\ \quad (n', ((\textsc{rx}, g), b, e, a)) \in \mathcal{E}(W') \end{array} \right\}$$

$$\text{where } g = \text{local} \Rightarrow \sqsupseteq = \sqsupseteq^{pub} \text{ and } g = \text{global} \Rightarrow \sqsupseteq = \sqsupseteq^{priv}$$

Fig. 7. Permission-based conditions

a safe register file (i.e. a register file in $\mathcal{R}$) and paired with a memory satisfying the world to become a safe configuration. Note that integers and non-executable capabilities (e.g. RO and E capabilities) are considered safe program counters because when they are plugged into a register file and paired with a memory, the execution will immediately fail, which is safe.

The *value relation* $\mathcal{V}$ defines when words are safe. We make the value relation as liberal as possible by considering what is the most we can allow an adversary to use a capability for without breaking the memory protocols. Non-capability data is always safe because it provides no authority. Capabilities give the authority to manipulate memory and potentially break memory protocols, so they need to satisfy certain conditions to be safe. In Fig. 7, we define such a condition for each kind of permission a capability can have.

For capabilities with read permission, the *readCond* ensures that it can only be used to read safe words, i.e. words in the value relation. To guarantee this, we require that the addressed memory is governed by a region $W(r)$ that imposes safety as a requirement on the values contained. This safety requirement is formulated in terms of a standard region $\iota^{pwl}_{b,e}$. The definition of that standard region is omitted for space reasons, but it simply requires all the words in the range $[b, e]$ to be safe, i.e. in the value relation. Requiring that $W(r) \overset{n}{\subsetapprox} \iota^{pwl}_{b,e}$ means that $W(r)$ must accept only safe values like $\iota^{pwl}_{b,e}$, but can be even more restrictive if desired. The read condition also takes into account the locality of the capability because, generally speaking, global capabilities should only depend on permanent regions. Concretely, we use the function *localityReg*(g, W), which projects out all active (non-revoked) regions when the locality g is local, but only the permanent regions when g is global. The definition of the standard region $\iota^{pwl}_{b,e}$ can be found in [14]; it makes use of the isomorphism from Theorem 1.

For a capability with write permission, *writeCond* must be satisfied for the capability's range of authority. An adversary can use such a capability to write any word they can get a hold of, and we can safely assume that they can only get a hold of safe words, so the region governing the relevant memory must allow any safe word to be written there. In order to make the logical relation as liberal as possible, we make this a lower bound of what the region may allow. For write capabilities, we also have to take into account the two flavours of write permissions: write and write-local. In the case of write-local capabilities, the region needs to allow (at least) any safe word to be written, but in the case of write capabilities, the capability cannot be used to write local capabilities, so the region only needs to allow safe non-local values. In the write condition, this is handled by parameterizing it with a region. For the write-local capabilities the write condition is applied with the standard region $\iota^{pwl}_{b,e}$ that we described previously. For the write capabilities we use a different standard region $\iota^{nwl}_{b,e}$ which requires that the words in $[b, e]$ are non-local and safe. As before, we use *localityReg* to pick an appropriate region based on the capability's locality. Finally, there is a technical requirement that the region must be *address-stratified*. Intuitively, this means that if a region accepts two memory segments, then it must also accept every memory segment "in between", that is every memory segment where each address contains a value from one of the two accepted memory segments. An interesting property of the write condition is that they prohibit global write-local capabilities which, as discussed in Sect. 3, is necessary for any safe use of local capabilities.

The conditions *enterCond* and *execCond* are very similar. Both require that the capability can be safely jumped to. However, executable capabilities can be updated to point anywhere in their range, so they must be safe as a program counter (in the $\mathcal{E}$-relation) no matter the current address. In contrast, enter capabilities are opaque and can only be used to jump to the address they point to. They also change permission when jumped to, so we require them to be safe as a program counter after the permission is changed to RX. Because the capabilities are not necessarily invoked immediately, this must be true in any future world, but it depends on the capability's locality which future worlds we consider. If it is global, then we require safety as a program counter in *private* future worlds (where temporary regions may be revoked). For local capabilities, it suffices to be safe in *public* future worlds, where temporary regions are still present.

In the technical appendix, we prove that safety of all values is preserved in public future worlds, and that safety of global values is also preserved in private future worlds:

Lemma 1 (Double monotonicity of value relation)

- *If* $W' \sqsupseteq^{pub} W$ *and* $(n, w) \in \mathcal{V}(W)$*, then* $(n, w) \in \mathcal{V}(W')$.
- *If* $W' \sqsupseteq^{priv} W$ *and* $(n, w) \in \mathcal{V}(W)$ *and* $w = ((perm, \text{global}), b, e, a)$ *(i.e. w is a global capability), then* $(n, w) \in \mathcal{V}(W')$.

4.3 Safety of the Capability Machine

With the logical relation defined, we can now state the fundamental theorem of our logical relation: a strong theorem that formalizes the guarantees offered by the capability machine. Essentially, it says a capability that only grants safe authority is capability safe as a program counter.

Theorem 2 (Fundamental theorem). *If one of the following holds:*

- $\mathit{perm} = \textsc{rx}$ *and* $(n, (b, e)) \in \mathit{readCond}(g)(W)$
- $\mathit{perm} = \textsc{rwx}$ *and* $(n, (b, e)) \in \mathit{readCond}(g)(W)$ *and* $(n, (b, e)) \in \mathit{writeCond}(\iota^{nwl}, g)(W)$
- $\mathit{perm} = \textsc{rwlx}$ *and* $(n, (b, e)) \in \mathit{readCond}(g)(W)$ *and* $(n, (b, e)) \in \mathit{writeCond}(\iota^{pwl}, g)(W)$,

then $(n, ((\mathit{perm}, g), b, e, a)) \in \mathcal{E}(W)$

The permission based conditions of Theorem 2 make sure that the capability only provides safe authority in which case the capability must be in the $\mathcal{E}$ relation, i.e. it can safely be used as a program counter in an otherwise safe register-file.

The Fundamental Theorem can be understood as a general expression of the guarantees offered by the capability machine, an instance of a general property called capability safety [11,12]. To understand this, consider that the theorem says the capability $((\mathit{perm}, g), b, e, a)$ is safe as a program counter, without any assumption about what instructions it actually points to (the only assumptions we have are about the read or write authority that it carries). As such, the theorem expresses the capability safety of the machine, which guarantees that *any* instruction is fine and will not be able to go beyond the authority of the values it has access to. We demonstrate this in Sect. 5 where Theorem 2 is used to reason about capabilities that point to arbitrary instructions. The relation between Theorem 2 and local-state encapsulation and control-flow correctness, will also be shown by example in Sect. 5 as the examples depend on these properties for correctness. See the technical appendix [14] for a detailed proof (by induction over the step-index n) of the theorem.

5 Examples

In this section, we demonstrate how our formalization of capability safety allows us to prove local-state encapsulation and control-flow correctness properties for challenging program examples. The security measures of Sect. 3 are deployed to ensure these properties. Since we are dealing with assembly language, there are many details to the formal treatment, and therefore we necessarily omit some details in the lemma statements. The examples may look deceivingly short, but it is because they use the macro instructions described in Sect. 3. The examples would be unintelligible without the macros, as each macro expands to multiple basic instructions. The interested reader can find all the technical details in the technical appendix [14].

```
f1: push 1                  f2: malloc r_l 1
    fetch r_1 adv               store r_l 1
    scall r_1([],[])            fetch r_1 adv
    pop r_1                     call r_1([],[r_l])
    assert r_1 1                assert r_l 1
    halt                        halt
```

Fig. 8. Two example programs that rely on local-state encapsulation. `f1` uses our stack-based calling convention. `f2` does not rely on a stack.

5.1 Encapsulation of Local State

`f1` and `f2` in Fig. 8 demonstrate the capability machine's encapsulation of local state. They are very similar: both store some local state, call an untrusted piece of code (*adv*), and then test whether the local state is unchanged. They differ in the way they do this. Program `f1` uses our stack-based calling convention (captured by `scall`) to call the adversary, so it can use the available stack to store its local state. On the other hand, `f2` uses malloc to allocate memory for its local state and uses an activation-record based calling convention (described in the technical appendix) to run the adversarial code.

For both programs, we can prove that if they are linked with an adversary, *adv*, that is allowed to allocate memory but has no other capabilities, then the assertion will never fail during executing (see Lemmas 2 and 3 below). The two examples also illustrate the versatility of the logical relation. The logical relation is not specific to any calling convention, so we can use it to reason about both programs, even though they use different calling conventions.

In order to formulate results about `f1` and `f2`, we need a way to observe whether the assertion fails. To this end, we assume they have access to a flag (an address in memory). If the assertion fails, then the flag is set to 1 and execution halts. The correctness lemma for `f1` then states:

Lemma 2. *Let*

$$\begin{array}{rlrl} c_{adv} & \overset{def}{=} ((\textsc{e}, \text{global}), \dots) & c_{stk} & \overset{def}{=} ((\textsc{rwlx}, \text{local}), \dots) \\ c_{f1} & \overset{def}{=} ((\textsc{rwx}, \text{global}), \dots) & c_{link} & \overset{def}{=} ((\textsc{ro}, \text{global}), \dots) \\ c_{malloc} & \overset{def}{=} ((\textsc{e}, \text{global}), \dots) & reg & \in \text{Reg} \end{array}$$

$$m \overset{def}{=} ms_{f1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame}$$

where each of the capabilities have an appropriate range of authority and pointer[2]*. Furthermore*

- ms_{f1} *contains* c_{link}, c_{flag} *and the code of* `f1`
- $ms_{flag}(flag) = 0$
- ms_{link} *contains* c_{adv} *and* c_{malloc}
- ms_{adv} *contains* c_{link} *and otherwise only instructions.*

If $(reg[\text{pc} \mapsto c_{f1}][r_{stk} \mapsto c_{stk}], m) \to^* (halted, m')$, *then* $m'(flag) = 0$

[2] These assumptions are kept intentionally vague for brevity. Full statements are in the technical appendix [14].

To prove Lemma 2, it suffices to show that the start configuration is safe (in the $\mathcal{O}$ relation) for a world with a permanent region that requires the assertion flag to be 0. By an anti-reduction lemma, it suffices to show that the configuration is safe after some reduction steps. We then use a general lemma for reasoning about `scall`, by which it suffices to show that (1) the configuration that `scall` will jump to is safe and (2) that the configuration just after `scall` is done cleaning up is safe. We use the Fundamental Theorem to reason about the unknown adversarial code, but notice that the adversary capability is an enter capability, which the Fundamental Theorem says nothing about. Luckily the enter capability becomes RX after the jump and then the Fundamental Theorem applies.

We have a similar lemma for `f2`:

Lemma 3. *Making similar assumptions about capabilities and linking as in Lemma 2 but assuming no stack pointer, if* $(reg[\mathrm{pc} \mapsto c_{f2}], m) \to^* (halted, m')$, *then* $m'(flag) = 0$.

5.2 Well-Bracketed Control-Flow

Using the stack-based calling convention of `scall`, we get well-bracketed control-flow. To illustrate this, we look at two example programs `f3` and `g1` in Fig. 9.

In `f3` there are two calls to an adversary and in order for the assertion in the middle to succeed, they need to be well-bracketed. If the adversary were able to store the return pointer from the first call and invoke it in the second call, then `f3` would have 2 on top of its stack and the assertion would fail. However, the security measures in Sect. 3 prevent this attack: specifically, the return pointer is local, so it can only be stored on the stack, but the part of the stack that is accessible to the adversary is cleared before the second invocation. In fact, the following lemma shows that there are also no other attacks that can break well-bracketedness of this example, i.e. the assertion never fails. It is similar to the two previous lemmas:

Lemma 4. *Making similar assumptions about capabilities and linking as in Lemma 2 if* $(reg[\mathrm{pc} \mapsto c_{f3}][r_{stk} \mapsto c_{stk}], m) \to^* (halted, m')$, *then* $m'(flag) = 0$.

The final example, `g1` with `f4`, is a faithful translation of a tricky example known from the literature (known as the awkward example) [13,20]. It consists of two parts, `g1` and `f4`. `g1` is a closure generator that generates closures with one variable x set to 0 in its environment and `f4` as the program (note we can omit some calling convention security measures because the stack is not used in the closure generator). `f4` expects one argument, a callback. It sets x to 0 and calls the callback. When it returns, it sets x to 1 and calls the callback a second time. When it returns again, it asserts x is 1 and returns. This example is more complicated than the previous ones because it involves a closure invoked by the adversary and an adversary callback invoked by us. As explained in Sect. 3, this means that we need to check (1) that the stack pointer that the closure receives from the adversary has write-local permission and (2) that the adversary callback is global.

```
g1:  malloc $r_2$ 1
     store $r_2$ 0
     move pc $r_3$
     lea $r_3$ $\mathit{offset}$
     crtcls $[(x, r_2)]$ $r_3$
     rclear $\mathrm{RegName} \setminus \{\mathrm{pc}, r_0, r_1\}$
     jmp $r_0$
f4:  reqglob $r_1$
     prepstk $r_{stk}$
     (continues in next column)

     (continued from previous column)
     store $x$ 0
     scall $r_1$([],$[r_0, r_1, r_{env}]$)
     store $x$ 1
     scall $r_1$([],$[r_0, r_{env}]$)
     load $r_1$ $x$
     assert $r_1$ 1
     mclear $r_{stk}$
     rclear $\mathrm{RegName} \setminus \{r_0, \mathrm{pc}\}$
     jmp $r_0$

f3:  push 1
     fetch $r_1$ $\mathit{adv}$
     scall $r_1$([],$[r_1]$)
     pop $r_2$
     assert $r_2$ 1
     push 2
     scall $r_1$([],[])
     halt
```

Fig. 9. Two programs that rely on well-bracketedness of `scall`s to function correctly. *offset* is the offset to `f4`.

To illustrate how subtle this program is, consider how an adversary could try to make the assertion fail. In the second callback an adversary can get to the first callback by invoking the closure one more time. If there were any way for the adversary to transfer the return pointer from the point where it reinvokes the closure to where the closure reinvokes the callback, then the assertion could be made to fail. Similarly, if there were any way for the adversary to store a stack pointer or trick the trusted code into preserving it across an invocation, the assertion can likely be made to fail too. However, our calling convention prevents any of this from happening, as we prove in the following lemma.

Lemma 5. *Let*

$$c_{adv} \overset{def}{=} ((\mathrm{RWX}, \mathrm{global}), \dots) \; c_{g1} \overset{def}{=} ((\mathrm{E}, \mathrm{global}), \dots)$$

and otherwise make assumptions about capabilities and linking similar to Lemma 2. Then if $(\mathit{reg}_0[\mathrm{pc} \mapsto c_{adv}][r_{stk} \mapsto c_{stk}][r_1 \mapsto c_{g1}], m) \rightarrow^* (\mathit{halted}, m')$, *then* $m'(\mathit{flag}) = 0$.

As explained in Sect. 3, the macro-instruction `reqglob` r_1 checks that the callback is global, essentially to make sure it is not allocated on the stack where it might contain old stack pointers or return pointers. Otherwise, the encapsulation of our local stack frame could be broken. In the proof of Lemma 5, this requirement shows up because we invoke the callback in a world that is only a private future world of the one where we received the callback, precisely because we have invalidated the adversary's local state (particularly their old stack and return capabilities). The callback is still valid in this private future world, but only because we know that it is global.

In Lemma 5 the order of control has been inverted compared to the previous lemmas. In this lemma, the adversary assumes control first with a capability for the closure creator `g1`. Consequently, we need to check that all arguments are safe to use and that we clean up before returning in the end. The inversion of control poses an interesting challenge when it comes to reasoning about the adversary's local state during the execution of `f4` and the callbacks where the adversary should not rely on the local state from before the call of `f4`. This is easily done by revoking all the temporary regions of the world given at the start of `f4`. However, when `f4` returns, the adversary is again allowed to rely on its old local state so we need to guarantee that the local state is unchanged. This is important because the return pointer that `f4` receives may be local, and the adversary is allowed to allocate the activation record on the stack (just like we do) so they can store and recover their old stack pointer after `f4` returns. By utilizing the reinstation mechanism of the future world relation as well as our knowledge of the future worlds used, we can construct a world in which the adversary's invariants are preserved. The details of this and the proofs of the other lemmas are found in the technical appendix [14].

6 Discussion

Calling Convention

Formulating Control Flow Correctness. While we claim that our calling convention enforces control-flow correctness, we do not prove a general theorem that shows this, because it is not clear what such a theorem should look like. Formulations in terms of a control-flow graph, like the one by Abadi et al. [2], do not take into account temporal properties, like the well-bracketedness that Example `g1` relies on. In fact, our examples show that our logical relation imply a stronger form of control-flow correctness than such formulations, although this is not made very explicit. As future work, we consider looking at a more explicit and useful way to formalize control-flow correctness. The idea would be to define a variant of our capability machine with call and return instructions and well-bracketed control flow built-in to the operational semantics, and then prove that compiling such code to our machine using our calling convention is fully abstract [21].

Performance and the Requirement for Stack Clearing. The additional security measures of the calling convention described in Sect. 3 impose an overhead compared to a calling convention without security guarantees. However, most of our security measures require only a few atomic checks or register clearings on boundary crossings between trusted code and adversary, which should produce an acceptable performance overhead. The only exception are the requirements for stack clearing that we have in two situations: when returning to the adversary and when invoking an adversary callback. As we have explained, we need to clear all of the stack that we are not using ourselves, not just the part that we have actually used. In other words, on every boundary cross between trusted code and adversary code, a potentially large region of memory must be cleared. We believe this is actually a common requirement for typical usage scenarios of local capabilities and capability machines like CHERI should consider to provide special support for this requirement, in the form of a highly-optimized instruction for erasing a large block of memory. Nevertheless, from a discussion with the designers of the CHERI capability machine, we gather that it is not immediately clear whether and how such a primitive could be implemented efficiently in the CHERI context.

Modularity. It is important that our calling convention is modular, i.e. we do not assume that our code is specially privileged w.r.t. the adversary, and they can apply the same measures to protect themselves from us as we do to protect ourselves from them. More concretely, the requirements we have on callbacks and return pointers received from the adversary are also satisfied by callbacks and return pointers that we pass to them. For example, our return pointers are local capabilities because they must point to memory where we can store the old stack pointer, but the adversary's return pointers are also allowed to be local. Adversary callbacks are required to be global but the callbacks we construct are allocated on the heap and also global.

Arguments and Local Capabilities. Local capabilities are a central part of the calling convention as they are used to construct stack and return pointers. The use of local capabilities for the calling convention unfortunately limits the extent to which local capabilities can be used for other things. Say we are using the calling convention and receive a local capability other than the stack and return pointer, then we need to be careful if we want to use it because it may be an alias to the stack pointer. That is, if we first push something to the stack and then write to the local capability, then we may be (tricked into) overwriting our own local state. The logical relation helps by telling us what we need to ascertain or check in such scenarios to guarantee safety and preserve our invariants, but such checks may be costly and it is not clear to us whether there are practical scenarios where this might be realistic.

We also need to be careful when we receive a capability from an adversary that we want to pass on to a different (instance of the) adversary. It turns out that the logical relation again tells us when this is safe. Namely, the logical relation says that we can only pass on safe arguments. For instance, when we receive a stack pointer from an adversary, then we may at some point want to pass on part of this stack pointer to, say, a callback. In order to do so, we need to make sure the stack pointer is safe which means that, if we have revoked temporary invariants, the stack must not directly or indirectly allow access to local values that we cannot guarantee safety of. When received from an adversary, we have to consider the contents of the stack unsafe, so before we pass it on, we have to clear it, or perform a dynamic safety analysis of the stack contents and anything it points to. Clearing everything is not always desirable and a dynamic safety analysis is hard to get right and potentially expensive.

In summary, the use of local capabilities for other things than stack and return pointers is likely only possible in very specific scenarios when using our calling convention. While this is unfortunate, it is not unheard of that processors have built-in constructs that are exclusively used for handling control flow, such as, for example, the call and return instructions that exist in some instruction sets.

Single Stack. A single stack is a good choice for the simple capability machine presented here, because it works well with higher-order functions. An alternative to a single stack would be to have a separate stack per component. The trouble with this approach is that, with multiple stacks and local stack pointers, it is not clear how components would retrieve their stack pointer upon invocation without compromising safety. A safe approach could be to have stack pointers stored by a central, trusted stack management component, but it is not clear how that could scale to large numbers of separate components. Handling large numbers of components is a requirement if we want to use capability machines to enforce encapsulation of, for example, every object in an object-oriented program or every closure in a functional program.

Logical Relation

Single Orthogonal Closure. The definitions of $\mathcal{E}$ and $\mathcal{V}$ in Fig. 6 apply a single orthogonal closure, a new variant of an existing pattern called biorthogonality. Biorthogonality is a pattern for defining logical relations [20,22] in terms of an observation relation of safe configurations (like we do). The idea is to define safe evaluation contexts as the set of contexts that produce safe observations when plugging safe values and define safe terms as the set of terms that can be plugged into safe evaluation contexts to produce safe observations. This is an alternative to more direct definitions where safe terms are defined as terms that evaluate to safe values. An advantage of biorthogonality is that it scales better to languages with control effects like call/cc. Our definitions can be seen as a variant of biorthogonality, where we take only a single orthogonal closure: we do not define safe evaluation contexts but immediately define safe terms as those that produce safe observations when plugged with safe values. This is natural because we model arbitrary assembly code that does not necessarily respect a particular calling convention: return pointers are in principle values like all others and there is no reason to treat them specially in the logical relation.

Interestingly, Hur and Dreyer [23] also use a step-indexed, Kripke logical relation for an assembly language (for reasoning about correct compilation from ML to assembly), but because they only model non-adversarial code that treats return pointers according to a particular calling convention, they can use standard biorthogonality rather than a single orthogonal closure like us.

Public/Private Future Worlds. A novel aspect of our logical relation is how we model the temporary, revokable nature of local capabilities using public/private future worlds. The main insight is that this special nature generalizes that of the syntactically-enforced unstorable status of evaluation contexts in lambda calculi without control effects (of which well-bracketed control flow is a consequence). To reason about code that relies on this (particularly, the original awkward example), Dreyer et al. [13] (DNB) formally capture the special status of evaluation contexts using Kripke worlds with public and private future world relations. Essentially, they allow relatedness of evaluation contexts to be monotone with respect to a weaker future world relation (public) than relatedness of values, formalizing the idea that it is safe to make temporary internal state modifications (private world transitions, which invalidate the continuation, but not other values) while an expression is performing internal steps, as long as the code returns to a stable state (i.e. transitions to a public future world of the original) before returning. We generalize this idea to reason about local capabilities: validity of local capabilities is allowed to be monotone with respect to a weaker future-world relation than other values, which we can exploit to distinguish between state changes that are always safe (public future worlds) and changes that are only valid if we clear all local capabilities (private future worlds). Our future world relations are similar to DNB's (for example, our proof of the awkward example uses exactly the same state transition system), but they turn up in an entirely different place in the logical relation: rather than using public future worlds for the special syntactic category of evaluation contexts, they are used in the value relation depending on the locality of the capability at hand. Additionally, our worlds are a bit more complex because, to allow local memory capabilities and write-local capabilities, they can contain (revokable) temporary regions that are only monotonous w.r.t. public future worlds, while DNB's worlds are entirely permanent.

Local Capabilities in High-Level Languages. We point out that local capabilities are quite similar to a feature proposed for the high-level language Scala: Osvald et al. [24]'s second-class or local values. They are a kind of values that can be provided to other code for immediate use without allowing them to be stored in a closure or reference for later use. We believe reasoning about such values will require techniques similar to what we provide for local capabilities.

7 Related Work

Finally, we summarize how our work relates to previous work. We do not repeat the work we discussed in Sect. 6.

Capability machines originate with Dennis and Van Horn [7] and we refer to Levy [25] and Watson et al. [9] for an overview of previous work. The capability machine formalized in Sect. 2 is a simple but representative model, modeled mainly after the M-Machine [6] (the enter pointers resemble the M-Machine's) and CHERI [9,10] (the memory and local capabilities resemble CHERI's). The latter is a recent and relatively mature capability machine, which combines capabilities with a virtual memory approach, in the interest of backwards compatibility and gradual adoption. As discussed, our local capabilities can cross module boundaries, contrary to what is enforced by CHERI's default CCall implementation.

Plenty of other papers enforce well-bracketed control flow at a low level, but most are restricted to preventing particular types of attacks and enforce only partial correctness of control flow. This includes particularly the line of work on *control-flow integrity* [2]. Those use a quite different attacker model than us: they assume an attacker that is not able to execute code, but can overwrite arbitrary data at any time during execution (to model buffer overflows). By checking the address of every indirect jump and using memory access control to prevent overwriting code, this work enforces what they call control-flow integrity, formalized as the property that every jump will follow a legal path in the control-flow graph. As discussed in Sect. 6, such a property ignores temporal properties and seems hard to use for reasoning.

More closely related to our work are papers that use a trusted stack manager and some form of memory isolation to enforce control-flow correctness as part of a secure compilation result [26,27]. Our work differs from theirs in that we use a different form of low-level security primitive (a capability machine with local capabilities rather than a machine with a primitive notion of compartments) and we do not use a trusted stack manager, but a decentralized calling convention based on local capabilities. Also, both prove a secure compilation result from a high-level language, which clearly implies a general form of control-flow correctness, while we define a logical relation that can be used to reason about specific programs that rely on well-bracketed control flow.

Our logical relation is a unary, step-indexed Kripke logical relation with recursive worlds [16,18,20,28], closely related to the one used by Devriese et al. [11] to formulate capability safety in a high-level JavaScript-like lambda calculus. Our Fundamental Theorem is similar to theirs and expresses capability safety of the capability machine. Because we are not interested in externally observable side-effects (like console output or memory access traces), we do not require their notion of effect parametricity. Our logical relation uses several ideas from previous work, like Kripke worlds with regions containing state transition systems [15], public/private future worlds [13] (see Sect. 6 for a discussion), and biorthogonality [20,23,29].

Swasey et al. [30] have recently developed a *logic*, OCPL, for verification of object capability patterns. The logic is based on Iris [31–33], a state of the art higher-order concurrent separation logic and is formalized in Coq, building on the Iris Proof Mode for Coq [34]. OCPL gives a more abstract and modular way of proving capability safety for a lambda-calculus (with concurrency) compared to the earlier work by Devriese et al. [11].

El-Korashy also defined a formal model of a capability machine, namely CHERI, and uses it to prove a compartmentalization result [35] (not implying control-flow correctness). He also adapts control-flow integrity (see above) to the machine and shows soundness, seemingly without relying on capabilities.

Acknowledgements. 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). Dominique Devriese holds a Postdoctoral fellowship from the Research Foundation Flanders (FWO).

References

1. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: Symposium on Operating Systems Principles, pp. 203–216. ACM (1993)
2. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity. In: Conference on Computer and Communications Security, pp. 340–353. ACM (2005)
3. Morrisett, G., Walker, D., Crary, K., Glew, N.: From system F to typed assembly language. ACM Trans. Program. Lang. Syst. **21**(3), 527–568 (1999)
4. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The Java Virtual Machine Specification. Pearson Education, London (2014)

5. Forrest, S., Somayaji, A., Ackley, D.H.: Building diverse computer systems. In: Hot Topics in Operating Systems, pp. 67–72, May 1997
6. Carter, N.P., Keckler, S.W., Dally, W.J.: Hardware support for fast capability-based addressing. In: Architectural Support for Programming Languages and Operating Systems, pp. 319–327. ACM (1994)
7. Dennis, J.B., Van Horn, E.C.: Programming semantics for multiprogrammed computations. Commun. ACM **9**(3), 143–155 (1966)
8. Shapiro, J.S., Smith, J.M., Farber, D.J.: EROS: a fast capability system. In: Symposium on Operating Systems Principles, SOSP 1999, pp. 170–185. ACM (1999)
9. Watson, R.N.M., Woodruff, J., Neumann, P.G., Moore, S.W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., Murdoch, S.J., Norton, R., Roe, M., Son, S., Vadera, M.: CHERI: a hybrid capability-system architecture for scalable software compartmentalization. In: IEEE Symposium on Security and Privacy, pp. 20–37 (2015)
10. Woodruff, J., Watson, R.N., Chisnall, D., Moore, S.W., Anderson, J., Davis, B., Laurie, B., Neumann, P.G., Norton, R., Roe, M.: The CHERI capability model: revisiting RISC in an age of risk. In: International Symposium on Computer Architecture, pp. 457–468. IEEE Press (2014)
11. Devriese, D., Birkedal, L., Piessens, F.: Reasoning about object capabilities using logical relations and effect parametricity. In: IEEE European Symposium on Security and Privacy. IEEE (2016)
12. Maffeis, S., Mitchell, J., Taly, A.: Object capabilities and isolation of untrusted web applications. In: S&P, pp. 125–140. IEEE (2010)
13. Dreyer, D., Neis, G., Birkedal, L.: The impact of higher-order state and control effects on local relational reasoning. J. Funct. Program. **22**(4–5), 477–528 (2012)
14. Skorstengaard, L., Devriese, D., Birkedal, L.: Reasoning about a machine with local capabilities: provably safe stack and return pointer management - technical appendix including proofs and details. Technical report, Department of Computer Science, Aarhus University (2018). https://cs.au.dk/~birke/papers/local-capabilities-conf-tr.pdf
15. Ahmed, A., Dreyer, D., Rossberg, A.: State-dependent representation independence. In: POPL, pp. 340–353. ACM (2009)
16. Ahmed, A.J.: Semantics of types for mutable state. Ph.D. thesis, Princeton University (2004)
17. Thamsborg, J., Birkedal, L.: A Kripke logical relation for effect-based program transformations. In: ICFP, pp. 445–456. ACM (2011)
18. Birkedal, L., Reus, B., Schwinghammer, J., Støvring, K., Thamsborg, J., Yang, H.: Step-indexed Kripke models over recursive worlds. In: POPL, pp. 119–132. ACM (2011)
19. Birkedal, L., Bizjak, A.: A Taste of Categorical Logic Tutorial Notes (2014). http://cs.au.dk/~birke/modures/tutorial/categorical-logic-tutorial-notes.pdf
20. Pitts, A.M., Stark, I.D.B.: Operational reasoning for functions with local state. In: Gordon, A.D., Pitts, A.M. (eds.) Higher Order Operational Techniques in Semantics, pp. 227–274. Cambridge University Press, New York (1998)
21. Abadi, M.: Protection in programming-language translations: mobile object systems. In: Demeyer, S., Bosch, J. (eds.) ECOOP 1998. LNCS, vol. 1543, p. 291. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49255-0_70
22. Krivine, J.L.: Classical logic, storage operators and second-order lambda-calculus. Ann. Pure and Appl. Log. **68**(1), 53–78 (1994)

23. Hur, C.K., Dreyer, D.: A Kripke logical relation between ML and assembly. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 133–146. ACM (2011)
24. Osvald, L., Essertel, G., Wu, X., Alayón, L.I.G., Rompf, T.: Gentrification gone too far? Affordable 2nd-class values for fun and (co-)effect. In: Object-Oriented Programming, Systems, Languages, and Applications, pp. 234–251. ACM (2016)
25. Levy, H.M.: Capability-Based Computer Systems, vol. 12. Digital Press, Bedford (1984)
26. Patrignani, M., Devriese, D., Piessens, F.: On modular and fully-abstract compilation. In: Computer Security Foundations Symposium (CSF), pp. 17–30, June 2016
27. Juglaret, Y., Hritcu, C., Amorim, A.A.D., Eng, B., Pierce, B.C.: Beyond good and evil: formalizing the security guarantees of compartmentalizing compilation. In: Computer Security Foundations Symposium (CSF), pp. 45–60, June 2016
28. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. Program. Lang. Syst. **23**(5), 657–683 (2001)
29. Benton, N., Hur, C.K.: Biorthogonality, step-indexing and compiler correctness. In: International Conference on Functional Programming, pp. 97–108. ACM (2009)
30. Swasey, D., Garg, D., Dreyer, D.: Robust and compositional verification of object capability patterns (2017, to appear)
31. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In: POPL, pp. 637–650 (2015)
32. Jung, R., Krebbers, R., Birkedal, L., Dreyer, D.: Higher-order ghost state. In: ICFP, pp. 256–269 (2016)
33. Krebbers, R., Jung, R., Bizjak, A., Jourdan, J.-H., Dreyer, D., Birkedal, L.: The essence of higher-order concurrent separation logic. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 696–723. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_26
34. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: POPL (2017)
35. El-Korashy, A.: A formal model for capability machines: an illustrative case study towards secure compilation to CHERI. Master's thesis, Saarland University, September 2016