Modular Denotational Semantics for Effects with Guarded Interaction Trees

DAN FRUMIN, University of Groningen, The Netherlands
AMIN TIMANY, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

We present *guarded interaction trees* — a structure and a fully formalized framework for representing higher-order computations with higher-order effects in Coq, inspired by domain theory and the recently proposed interaction trees. We also present an accompanying separation logic for reasoning about guarded interaction trees. To demonstrate that guarded interaction trees provide a convenient domain for interpreting higher-order languages with effects, we define an interpretation of a PCF-like language with effects and show that this interpretation is sound and computationally adequate; we prove the latter using a logical relation defined using the separation logic. Guarded interaction trees also allow us to combine different effects and reason about them modularly. To illustrate this point, we give a modular proof of type soundness of cross-language interactions for safe interoperability of different higher-order languages with different effects. All results in the paper are formalized in Coq using the Iris logic over guarded type theory.

CCS Concepts: • **Theory of computation → Program semantics**; *Logic and verification*; • **Software and its engineering** → *Software libraries and repositories.*

Additional Key Words and Phrases: Coq, Iris, denotational semantics, logical relations

ACM Reference Format:
Dan Frumin, Amin Timany, and Lars Birkedal. 2024. Modular Denotational Semantics for Effects with Guarded Interaction Trees. *Proc. ACM Program. Lang.* 8, POPL, Article 12 (January 2024), 30 pages. https://doi.org/10.1145/3632854

1 INTRODUCTION

Interaction trees [Xia et al. 2019] are a recently proposed formalism for representing and reasoning about (possibly) non-terminating programs with side effects in Coq (a terminating type theory without effects). Since its inception, interaction trees have been applied, including but not limited, to specifying and verifying network servers [Koh et al. 2019; Zhang et al. 2021], semantics of LLVM [Zakowski et al. 2021], semantics of a language for robotics [Ye et al. 2022], non-interference [Silver et al. 2023], and verification of concurrent objects with transactional memory [Lesani et al. 2022].

The introduction of interaction trees was motivated by a desire to simplify mechanized formalizations of interactive, effectful, non-terminating computations and the developers of the ITrees library argued that ITrees can represent computations in a way which is more *modular* than representations based on operational semantics and *executable* (in contrast to earlier representations based on traces represented as predicates on events). In particular, the idea is that interaction trees can be used to give *denotational semantics* to programming languages and thus allow one to abstract away from syntactic details and reuse meta-language features such as function composition so as

Authors' addresses: Dan Frumin, d.frumin@rug.nl, University of Groningen, The Netherlands; Amin Timany, Aarhus University, Denmark, timany@cs.au.dk; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

ACM 2475-1421/2024/1-ART12
https://doi.org/10.1145/3632854

to obtain more robust mechanizations. And, indeed, the applications mentioned above demonstrate that interaction trees work well for giving semantics to first-order programming languages with first-order effects.

The challenge we address in this paper is that interaction trees cannot easily be used as a model of *higher-order* programming languages with *higher-order effects*, which, of course, limits the applicability of interaction trees. Indeed, the ease of use of interaction trees is enabled, in part, by two restrictions imposed on the computations represented by the interaction trees: the computations must be first-order, and the effects that the computation performs must be first-order as well. With those restrictions, the type of interaction trees forms a monad, which allows one to compose the represented computations and reason about them modularly. (In principle, one could represent higher-order computations by means of closures in interaction trees, but that would defeat the purpose of interaction trees and force one to reason about syntactic representations, which interaction trees otherwise relieves one from.) To understand the limitations to first-order programs and first-order effects, we call to mind the definition of interaction trees.

Interaction trees are possibly infinite trees with two types of branching. The first type of branching represents a "delayed" computation (similar to that of the delay monad), or a computation performing a silent step. The second type of branching represents a computation that performs an effect; different results of the effect lead to different branches. Interaction trees are formalized as coinductive types in Coq, allowing one to leverage existing infrastructure for coinductive programs and proofs:

```
CoInductive itree (E : Type -> Type) (R : Type) :=
| Ret : R -> itree E R
| Tau : itree E R -> itree E R
| Vis {A : Type} : E A -> (A -> itree E R) -> itree E R
```

Now the point is that if we wish to represent higher-order computations, then we cannot simply add a constructor `Fun : (itree E R -> itree E R) -> itree E R`, as the resulting recursive type would have negative occurrences of the recursive variable (the `itree E R` on the left of the first arrow). Similarly, if we want to support computations with higher-order effects, i.e., the result of an effect is an interaction tree itself, we run into the same problems with positivity. For example, in the following potential signature for a higher-order effect, the parameter `test` occurs in a negative position:

```
Inductive test : Type -> Type :=
| T : nat -> test (itree test unit).
```

Guarded interaction trees: Iris and guarded type theory. Our *goal* is to address the challenge of extending interaction trees to allow for higher-order computations and higher-order effects, in such a way that we retain some of the advantages of interaction trees; in particular we wish to obtain a representation with which we can work efficiently in Coq. From the discussion above, it is clear that a way forward is to work in a setting that allows to solve mixed-variance recursive domain equations. There are several possible choices for such a setting, including classical Scott domain theory [Scott 1976; Smyth and Plotkin 1982] and guarded type theory [Birkedal et al. 2012, 2010]. We choose to use the latter since this choice allows us to leverage the Iris program logic framework in Coq and thence obtain an efficient environment in which we can work efficiently and formally in Coq.

Thus in this paper we introduce *guarded interaction trees*, which are formally defined in guarded type theory as a solution to a guarded recursive domain equation, and we show how guarded interaction trees can be used to represent higher-order computations and higher-order effects. Moreover, we demonstrate how we can retain some of the benefits of interaction trees, in particular modularity with respect to effects and ease of use in Coq. The extension to higher-order computations and effects does come with a certain price, in that we need to reason about guardedness, but we believe the use of Iris alleviates this.

Our Coq formalization is available online at

https://github.com/logsem/gitrees/tree/popl24.

(tag `popl24` in the Git repository)

Contributions. In this paper we present the following contributions, all formalized as part of our extensible and adaptable Coq formalization:

(1) We present *guarded interaction trees*, describe the associated recursion principle, and demonstrate how to write combinators to program with guarded interaction trees (Section 3).
(2) We describe a way of *reifying* effects in the guarded interaction trees, and the reduction semantics (Section 4).
(3) We show how to give a model of a higher-order programming language with general recursion and effects in guarded interaction trees, and show that the model is sound (Section 5).
(4) We build a separation logic (a program logic) on top of guarded interaction trees, allowing us to reason about their behavior (Section 6).
(5) We use the separation logic to show that the model that we construct in Section 5 satisfies computational adequacy (Section 7).
(6) We demonstrate how multiple different effects can be combined in guarded interaction trees, and how the separation logic is used to reason about the effects locally (Section 8).
(7) Finally, we utilize the results above, and use guarded interaction trees to show type safety of cross-language interactions for safe interoperability of languages with different effects (Section 9).

We discuss related work in Section 10. Before we present our results, we briefly go over some preliminaries about the setting that we are working in.

2 IRIS LOGIC OVER GUARDED TYPE THEORY

In this section we describe the Iris logic, in which we shall define and work with guarded interaction trees. Our treatment is brief since Iris has been described in many other papers and we are just using a small extension of the usual presentation; we refer the reader to the literature on Iris [Jung et al. 2018] and guarded type theory [Birkedal et al. 2012] for more details.

Iris is usually presented as a separation logic over a simple type theory.The model of Iris, however, models a richer type theory and in this paper we are going make use of that and consider Iris over a guarded type theory with (1) a modicum of dependent type theory, and (2) the ability to define guarded recursive types. Both of these features are supported by the existing Coq implementation of Iris and the associated Iris proof mode [Krebbers et al. 2017b].

Note that since we are working formally in Iris in Coq, there are two logical levels at play: the statements and proofs at the Coq level (which we refer to as the meta-logic level, or as the meta level), and the statements and proofs at the level of Iris (which we refer to as the logic level).

We recall the grammar of Iris in Figure 1; the syntax consists of types, terms, and propositions. Most of the grammar is standard for higher-order intuitionistic logic, with the parts related to guarded recursion highlighted in blue. As usual in higher-order logic, we have a well-typedness judgment $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau$ stating that the term t has type τ, under the assumption that the variables x_i have types τ_i. In the grammar for types, $\mathtt{I}$ ranges over so-called discrete types, which are meta-level types embedded into the types of Iris. Note that types include dependent types over discrete types. While we have not shown it in the grammar, we can also form types as solutions to

$$\begin{aligned}
\tau &::= \mathsf{iProp} \mid 0 \mid \mathbf{1} \mid \mathbb{B} \mid \mathsf{Nat} \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau \mid \blacktriangleright\tau \mid I \mid \Sigma_{i \in I}\tau_i \mid \Pi_{i \in I}\tau_i \mid \ldots \\
t &::= x \in \mathit{Var} \mid F(t_1, \ldots, t_n) \mid \mathsf{abort}\ t \mid () \mid (t, t) \mid \pi_i\ t \mid \lambda x : \tau.\, t \mid t(t) \mid \\
&\qquad \mathsf{inj}_i\ t \mid \mathsf{match}\ t\ \mathsf{with}\ \mathsf{inj}_1\ x.\, t \mid \mathsf{inj}_2\ x.\, t\ \mathsf{end} \mid \mathsf{next}(t) \mid \mathit{fix}_\tau \\
P &::= \mathsf{False} \mid \mathsf{True} \mid t =_\tau t \mid P \land P \mid P \lor P \mid P \to P \mid \exists x : \tau.\, P \mid \forall x : \tau.\, P \mid \triangleright P \mid \mu x : \tau.\, P
\end{aligned}$$

Fig. 1. Grammar for the base logic.

guarded recursive domain equations, i.e., type equations where the recursive occurence of the type being defined is guarded under the $\blacktriangleright$ type modality. Such types are defined up to isomorphism; we will see an example shortly: the type of guarded interaction trees will be such a recursive type and will be introduced in the following section. A useful semantic intuition for the types of Iris is that they denote (certain kinds of) time-indexed sets, i.e., families of sets indexed over natural numbers. At time step $n > 0$, the later type $\blacktriangleright\tau$ consists of the elements of τ at a later time step, i.e., at $n - 1$. At time step $n = 0$, the type $\blacktriangleright\tau$ is a singleton set. Intuitively, guarded recursive types exist because to understand what a guarded recusive type is at time step n, one only needs to understand what it is at $n - 1$, since the recursion is guarded.

Elements of $\blacktriangleright\tau$ can be constructed from elements of τ, using the next constructor, and we can form fixed points for guarded endo-functions:

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{next}(t) : \blacktriangleright\tau} \qquad\qquad \Gamma \vdash \mathit{fix}_\tau : (\blacktriangleright\tau \to \tau) \to \tau$$

The $\blacktriangleright$ type former is functorial and we write $\blacktriangleright f : \blacktriangleright\tau_1 \to \blacktriangleright\tau_2$ for its action on terms $f : \tau_1 \to \tau_2$. The fixed point satisfies the equation $\mathit{fix}_\tau(f) = f(\mathsf{next}(\mathit{fix}_\tau(f)))$.

For propositions P : iProp we also have the provability judgment $\Gamma \mid P \vdash Q$ stating that Q is derivable from P in the typing context $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$. The rules corresponding to the intuitionistic fragment are standard. Here we present the rules concerning the guarded part of the logic.

On the level of propositions, we have a(nother) *later modality* $\triangleright$. This is the later modality most users of Iris are already familiar with. It is related to the later modality on types in that $\triangleright(t =_\tau t') \dashv\vdash \mathsf{next}(t) =_{\blacktriangleright\tau} \mathsf{next}(t')$. We recall that $\triangleright$ can be used to define guarded recursive predicates and and that it supports reasoning via Löb induction:

$$\frac{\Gamma \vdash P : \mathsf{iProp}}{\Gamma \vdash \triangleright P : \mathsf{iProp}} \qquad \frac{\Gamma, x : \tau \vdash P : \mathsf{iProp} \qquad x \text{ is guarded in } P}{\Gamma \vdash \mu x : \tau.\, P : \mathsf{iProp}} \qquad \frac{\Gamma \mid \triangleright P \vdash P}{\Gamma \mid \mathsf{True} \vdash P}$$

Iris also includes separation logic connectives; we recall those later, when we need them, in Section 6.

If P is a proposition that consists only of intuitionistic logic connectives without $\triangleright$, then we can interpret it both as a meta-level proposition (i.e. a Coq proposition), and as an Iris proposition. For such propositions we have the following result, connecting Iris with the meta-level:

Theorem 2.1 (Iris Adequacy). *Let P be a proposition containing only intuitionistic connectives. Then, if True $\vdash P$ is derivable in Iris, then P also holds at the meta-level.*

3 GUARDED INTERACTION TREES

The type of *guarded interaction trees* (or *GITrees* for short) $\mathbf{IT}_E(A)$ is defined for a ground type A and an effect signature E, as we explain below. In Figure 2 $\mathbf{IT}_E(A)$ is written down as a guarded

$$\begin{aligned}
\text{guarded type } \mathbf{IT}_E(A) = {} & \mathsf{Ret} : A \to \mathbf{IT}_E \\
& \mid \mathsf{Fun} : \blacktriangleright(\mathbf{IT}_E(A) \to \mathbf{IT}_E(A)) \to \mathbf{IT}_E(A) \\
& \mid \mathsf{Err} : \mathtt{Error} \to \mathbf{IT}_E(A) \\
& \mid \mathsf{Tau} : \blacktriangleright\mathbf{IT}_E(A) \to \mathbf{IT}_E(A) \\
& \mid \mathsf{Vis} : \prod_{i \in I} \big(\mathit{Ins}_{\mathtt{i}}(\blacktriangleright\mathbf{IT}_E(A)) \times (\mathit{Outs}_{\mathtt{i}}(\blacktriangleright\mathbf{IT}_E(A)) \to \blacktriangleright\mathbf{IT}_E(A))\big) \to \mathbf{IT}_E(A)
\end{aligned}$$

Fig. 2. Guarded datatype of interaction trees.

datatype.[1] The first constructor Ret says that any element a of the ground type A can be associated with a "terminated" guarded interaction tree $\mathsf{Ret}(a)$. The second constructor Fun says that functions are also guarded interaction trees, and it is this constructor that allows us to model higher-order computations. Since the function constructor contains a negative occurrence of $\mathbf{IT}_E(A)$ in its argument, we must put it under a $\blacktriangleright$. The third constructor Err represents an error state, or a stuck computation, which we take from some predefined set $\mathtt{Error}$ of errors. We assume that it contains at least one element $\mathit{RunTime} \in \mathtt{Error}$ representing a generic run-time error. The fourth constructor Tau denotes a delayed computation, or a computation that is available "later". We also write $\mathsf{Tick} : \mathbf{IT}_E(A) \to \mathbf{IT}_E(A)$ for the composition $\mathsf{Tau} \circ \mathsf{next}$. Then the term $\mathsf{Tick}(\alpha)$ represents a guarded interaction tree that takes a silent step to α. It satisfies the following rule for equality: $\mathsf{Tick}(\alpha) = \mathsf{Tick}(\beta) \dashv\vdash \triangleright(\alpha = \beta)$.

Finally, the last constructor Vis allows us to model effects. The possible effects are given by the signature $E = (\mathtt{I}, \mathit{Ins}_{-}, \mathit{Outs}_{-})$, where $\mathtt{I}$ is an indexing set on the meta-level (i.e. a set of operation names), and *Ins* and *Outs* are functors determining the arities of the operations. That is $\mathit{Ins}_{\mathtt{i}}, \mathit{Outs}_{\mathtt{i}} : \mathit{Type} \to \mathit{Type}$ for $\mathtt{i} \in \mathtt{I}$. The *Type* argument in $\mathit{Ins}_{\mathtt{i}}$ and $\mathit{Outs}_{\mathtt{i}}$ is instantiated with $\mathbf{IT}_E$ itself, and is used for giving signatures to higher-order effects. With this, the first argument to $\mathsf{Vis}_{\mathtt{i}}$ is then the input for the operation, and the second parameter is a continuation which, given an arbitrary output of the operations, produces the remainder of the computation. One way to visualize this is to think of $\mathsf{Vis}_{\mathtt{i}}$ as a node in the tree, with the annotation $\mathit{Ins}_{\mathtt{i}}$ and having $\mathit{Outs}_{\mathtt{i}}$ many branches.

We refer to guarded interaction trees $\mathsf{Ret}(a)$ and $\mathsf{Fun}(f)$ as GITree values, and write $\mathbf{IT}^v_E(A) \subseteq \mathbf{IT}_E(A)$ for the set of values. When quantifying over an indexing set, we implicitly coerce E to $\mathtt{I}$, i.e. we write $\mathtt{i} \in E$ to mean $\mathtt{i} \in E.\mathtt{I}$. Similarly we write *Ins* for *E.Ins* and *Outs* for *E.Outs* when the signature E is clear from the context. When the signature E is obvious or unimportant we simply write $\mathbf{IT}(A)$ for $\mathbf{IT}_E(A)$ and $\mathbf{IT}^v(A)$ for $\mathbf{IT}^v_E(A)$.

Let us demonstrate the syntax of guarded interaction trees with some running examples of effects.

Example 3.1 (Input/output on a tape). Suppose we want to model two effectful operations, for reading a number from STDIN and for writing an output on STDOUT. We will model them as

[1]Formally, the datatype is given by a solution to a recursive equation, which we examine in Section 3.1. But it is convenient to think of $\mathbf{IT}_E(A)$ as a recursive datatype in which every recursive occurrence is behind a $\blacktriangleright$.

guarded interaction trees $\mathbf{IT}_{E_{io}}(\mathbf{1} + \mathsf{Nat})$, where $\mathbf{1} = \{()\}$ is the unit type and

$$\begin{aligned} E_{io} &\triangleq \{\texttt{input}, \texttt{output}\} \\ \mathit{Ins}_{\texttt{input}}(X) &\triangleq \mathbf{1} & \mathit{Outs}_{\texttt{input}}(X) &\triangleq \mathsf{Nat} \\ \mathit{Ins}_{\texttt{output}}(X) &\triangleq \mathsf{Nat} & \mathit{Outs}_{\texttt{ouput}}(X) &\triangleq \mathbf{1} \end{aligned}$$

We write Input and Output(n) for the GITrees

$$\mathsf{Input} \triangleq \mathsf{Vis}_{\texttt{input}}((), \lambda n.\, \mathsf{next}(\mathsf{Ret}(\mathsf{inr}(n)))) \quad \mathsf{Output}(n) \triangleq \mathsf{Vis}_{\texttt{output}}(n, \lambda x.\, \mathsf{next}(\mathsf{Ret}(\mathsf{inl}(()))))$$

Here we use $\mathsf{inl}(()) : \mathbf{1} + \mathsf{Nat}$ as a "dummy" value, since we do not care about the return value of Output.

The operations Input and Output above are represented as GITrees $\mathbf{IT}_{E_{io}}(\mathbf{1} + \mathsf{Nat})$. However, the exact ground type is not important, as long as it contains the unit $\mathbf{1}$ and the natural numbers Nat. As such, we assume that we can write down operations like Input and Output as GITrees $\mathbf{IT}_{E_{io}}(A)$ where $A \simeq \mathbf{1} + \mathsf{Nat} + B$ for some type B. We return again to this point in Section 8, but for now we assume that we always pick a ground type A that is "large enough" to represent all the ground values that we need.

Example 3.2 (Higher-order store). We can model higher-order store with the following signature.

$$\begin{aligned} E_{store} &\triangleq \{\texttt{alloc}, \texttt{read}, \texttt{write}, \texttt{dealloc}\} \\ \mathit{Ins}_{\texttt{alloc}}(X) &\triangleq X & \mathit{Outs}_{\texttt{alloc}}(X) &\triangleq \mathit{Loc} \\ \mathit{Ins}_{\texttt{read}}(X) &\triangleq \mathit{Loc} & \mathit{Outs}_{\texttt{read}}(X) &\triangleq X \\ \mathit{Ins}_{\texttt{write}}(X) &\triangleq \mathit{Loc} \times X & \mathit{Outs}_{\texttt{write}}(X) &\triangleq \mathbf{1} \\ \mathit{Ins}_{\texttt{dealloc}}(X) &\triangleq \mathit{Loc} & \mathit{Outs}_{\texttt{dealloc}}(X) &\triangleq \mathbf{1} \end{aligned}$$

where *Loc* is a countable type of locations/pointers. We write Alloc, Read, Write, and Dealloc for the following GITrees:

$$\begin{aligned} \mathsf{Alloc}(\alpha : \mathbf{IT}(A), k : \mathit{Loc} \to \mathbf{IT}(A)) &\triangleq \mathsf{Vis}_{\texttt{alloc}}(\mathsf{next}(\alpha), \mathsf{next} \circ k) \\ \mathsf{Read}(\ell : \mathit{Loc}) &\triangleq \mathsf{Vis}_{\texttt{read}}(\ell, \lambda x.\, x) \\ \mathsf{Write}(\ell : \mathit{Loc}, \alpha : \mathbf{IT}(A)) &\triangleq \mathsf{Vis}_{\texttt{write}}((\ell, \mathsf{next}(\alpha)), \lambda x.\, \mathsf{next}(\mathsf{Ret}(\mathsf{inj}(())))) \\ \mathsf{Dealloc}(\ell : \mathit{Loc}) &\triangleq \mathsf{Vis}_{\texttt{dealloc}}(\ell, \lambda x.\, \mathsf{next}(\mathsf{Ret}(\mathsf{inj}(())))) \end{aligned}$$

Here we assume that the ground type A is isomorphic to $\mathbf{1} + B$ for some B, with the injection $\mathsf{inj} : \mathbf{1} \to A$.

3.1 Recursion Principle for Guarded Interaction Trees

In order to write programs that eliminate GITrees, i.e. programs of type $\mathbf{IT}_E(A) \to P$, we need to come up with a suitable recursion principle. Recursion principles for inductive datatypes usually follow from the initiality principles of the defined datatypes. However, the type of GITrees is not purely inductive, as it has mixed-variance recursive occurrences, and the corresponding recursion principle should reflect that. To understand the necessary recursion principle we need to understand first how the GITrees are defined. The definition at the beginning of this section presents GITrees as a guarded datatype, but how should such a datatype be constructed? In the type theory, the type $\mathbf{IT}_E(A)$ is given as the solution to the following guarded equation:

$$\begin{aligned} \mathbf{IT}_E(A) \simeq{} & A + \blacktriangleright[\mathbf{IT}_E(A) \to \mathbf{IT}_E(A)] + \mathsf{Error} + \blacktriangleright\mathbf{IT}_E(A) + \\ & \Sigma_{i \in E}\big(\mathit{Ins}_i(\blacktriangleright\mathbf{IT}_E(A)) \times (\mathit{Outs}_i(\blacktriangleright\mathbf{IT}_E(A)) \to \blacktriangleright\mathbf{IT}_E(A))\big) \end{aligned} \tag{1}$$

The isomorphism is witnessed by the pair of functions $(\mathit{unfold}, \mathit{fold})$, and the constructors we presented at the beginning of the section are compositions of injections and *fold*. Since Equation (1) contains recursive occurrences with mixed variance, we cannot use the usual recursion principle for inductive data type. Instead, we employ a mixed initial-algebra/final-coalgebra principle, following [Freyd 1991; Pitts 1996]. To understand it better, we first write out the bi-functor, where the fixed point corresponds to the type of GITrees:[2]

$$F(X, Y) \simeq A + \blacktriangleright[X \to Y] + \mathit{Error} + \blacktriangleright Y + \Sigma_{i \in E}\big(\mathit{Ins}_{\mathtt{i}}(\blacktriangleright Y) \times (\mathit{Outs}_{\mathtt{i}}(\blacktriangleright X) \to \blacktriangleright Y)\big)$$

Here the bi-functor $F(-,-)$ is contravariant in the first argument and covariant in the second one. The bi-algebra corresponding to the type of GITrees is given by the $(\mathit{fold}, \mathit{unfold})$ pair:

$$\begin{tikzcd} F(\mathbf{IT}, \mathbf{IT}) \arrow[r, shift right, "\mathit{fold}"'] & \mathbf{IT} \arrow[l, shift right, "\mathit{unfold}"'] \end{tikzcd}$$

where we write $\mathbf{IT}$ as a shorthand for $\mathbf{IT}_E(A)$. The recursion principle that we are looking for then states that this bi-algebra is both initial and terminal. That is, for any other bi-algebra (P, f, g) we have unique maps h and k such that the following diagram commutes:

$$\begin{tikzcd} F(P, P) \arrow[rr, shift left, "F(k,h)"] \arrow[d, shift right, "f"'] & & F(\mathbf{IT}, \mathbf{IT}) \arrow[ll, shift left, "F(h,k)"] \arrow[d, shift right, "\mathit{fold}"'] \\ P \arrow[u, shift right, "g"'] \arrow[rr, shift left, "h"] & & \mathbf{IT} \arrow[u, shift right, "\mathit{unfold}"'] \arrow[ll, shift left, "k"] \end{tikzcd}$$

That is, in order to construct a function $k : \mathbf{IT} \to P$, one has to provide the "unfolding" $P \to F(P, P)$, as well as functions $A \to P$, $\mathtt{Error} \to P$, $\blacktriangleright P \to P$, $\blacktriangleright(P \to P) \to P$, and $\prod_{\mathtt{i} \in E} \mathit{Ins}_{\mathtt{i}}(\blacktriangleright P) \to (\mathit{Outs}_{\mathtt{i}}(\blacktriangleright P) \to \blacktriangleright P) \to P$. This alone would allow us to *iterate* over GITrees. However, we would run intro trouble if we wish to write a primitive-recursive style function. For example, we might wish to write a destructor function k such that $k(\alpha)$ returns α if α itself is a function $\mathtt{Fun}(f)$, and $\mathtt{Err}(\mathit{RunTime})$ otherwise. We cannot do so with the scheme outlined above, since in the recursive call we don't have access to the original argument, only to the result of applying recursion to the argument. This is similar to how the iteration scheme $B \to (B \to B) \to \mathbb{N} \to B$ for natural numbers does not allow us to (easily) write the predecessor function $p : \mathbb{N} \to \mathbb{N}$ satisfying $p(0) = 0$ and $p(n + 1) = n$ if we pick $B = \mathbb{N}$.

Recursion from iteration on inductive types. Let us then look at how to solve the issue of defining primitive recursive functions on inductive types using initiality. Suppose the function $p : \mathbb{N} \to B$ that we want to construct is defined by equations $p(0) = p_1$ and $p(n + 1) = p_2(n, p(n))$. Then we can obtain this function p using the following trick: instead of eliminating $\mathbb{N}$ into B using initiality, we eliminate it into $\mathbb{N} \times B$, in such a way that the induced map $\mathbb{N} \to \mathbb{N} \times B$ is the identity on the first component. More concretely, suppose we have maps $p_1 : 1 \to B$ and $p_2 : \mathbb{N} \times B \to B$, forming together the equations for primitive recursion. Then we construct an $\mathbb{N}$-algebra over $\mathbb{N} \times B$ as

$$1 + (\mathbb{N} \times B) \xrightarrow{[\langle 0, p_1 \rangle, \langle S, p_2 \rangle]} \mathbb{N} \times B$$

where $S : \mathbb{N} \to \mathbb{N}$ is the successor function. The initiality of $\mathbb{N}$ will then induce the unique map $p : \mathbb{N} \to \mathbb{N} \times B$, which, when composed with projection $\mathbb{N} \times B \to B$, determines the recursive function given by the clauses p_1 and p_2.

[2]For a detailed category-theoretic treatment, see [Birkedal et al. 2010].

Recursion/corecursion from mixed-variance types. Dually, for coinductive datatypes we can obtain a form of primitive coinduction from coiteration by using coproducts. In our case we have a datatype with mixed variance, and we use coproducts for the negative occurrences and products for the positive ones. That is, in order to eliminate $\mathbf{IT}$ into a type P we will assume an unfolding $P \to F(P, P)$, and a folding $F(\mathbf{IT} + P, \mathbf{IT} \times P) \to P$. More concretely:

Definition 3.3 (Recursion/corecursion principle). In order to define a pair of maps $\begin{tikzcd} P \arrow[r, shift left, "h"] & \mathbf{IT} \arrow[l, shift left, "k"] \end{tikzcd}$, one has to define maps

- $h_u : P \to A + \blacktriangleright[P \to P] + \mathit{Error} + \blacktriangleright P + \Sigma_{i \in E}\big(\mathit{Ins}_{\mathtt{i}}(\blacktriangleright P) \times (\mathit{Outs}_{\mathtt{i}}(\blacktriangleright P) \to \blacktriangleright P)\big)$;
- $k_{\mathsf{Ret}} : A \to P$;
- $k_{\mathsf{Fun}} : \blacktriangleright\big((\mathbf{IT} + P) \to (\mathbf{IT} \times P)\big) \to P$;
- $k_{\mathsf{Err}} : \mathtt{Error} \to P$;
- $k_{\mathsf{Tau}} : \blacktriangleright(\mathbf{IT} \times P) \to P$;
- $k_{\mathsf{Vis}} : \prod_{\mathtt{i}:E} \mathit{Ins}_{\mathtt{i}}(\blacktriangleright(\mathbf{IT} \times P)) \to (\mathit{Outs}_{\mathtt{i}}(\blacktriangleright(\mathbf{IT} + P)) \to \blacktriangleright(\mathbf{IT} \times P)) \to P$.

The resulting maps (h, k) will then satisfy the following *computational rules*:

- $k(\mathsf{Ret}(a)) = k_{\mathsf{Ret}}(a)$;
- $k(\mathsf{Fun}(f)) = k_{\mathsf{Fun}}(\blacktriangleright s(f))$ where $s(f) = \langle \mathsf{id}, k\rangle \circ f \circ [\mathsf{id}, h]$;
- $k(\mathsf{Err}(e)) = k_{\mathsf{Err}}(e)$;
- $k(\mathsf{Tau}(t)) = k_{\mathsf{Tau}}\big(\blacktriangleright\langle \mathsf{id}, k\rangle(t)\big)$;
- $k(\mathsf{Vis}_{\mathtt{i}}(x, k)) = k_{\mathsf{Vis}}(\mathtt{i},\ \mathit{Ins}_{\mathtt{i}}(\blacktriangleright\langle \mathsf{id}, k\rangle)(x),\ \blacktriangleright\langle \mathsf{id}, k\rangle \circ k \circ \mathit{Outs}_{\mathtt{i}}(\blacktriangleright[\mathsf{id}, h]))$;
- plus equations for h.

(Recall that we write $\blacktriangleright s : \blacktriangleright A \to \blacktriangleright B$ for a function $s : A \to B$.)

This recursion/corecursion principle is constructed using guarded recursion, and can be used to define a large variety of combinators. For example, we can write a generalization of the aforementioned function k that returns its argument, if the argument is a function, and returns an error otherwise.

Using the recursion principle we can define a function $\mathsf{get\_fun}(\alpha : \mathbf{IT}, f : \blacktriangleright(\mathbf{IT} \to \mathbf{IT}) \to \mathbf{IT})$ satisfying the computational rules

- $\mathsf{get\_fun}(\mathsf{Ret}(a), f) = \mathsf{Err}(\mathit{RunTime})$;
- $\mathsf{get\_fun}(\mathsf{Fun}(g), f) = f(g)$;
- $\mathsf{get\_fun}(\mathsf{Err}(e), f) = \mathsf{Err}(e)$;
- $\mathsf{get\_fun}(\mathsf{Tau}(t), f) = \mathsf{Tau}(\blacktriangleright\mathsf{get\_fun}(t, f))$ and $\mathsf{get\_fun}(\mathsf{Tick}(\alpha), f) = \mathsf{Tick}(\mathsf{get\_fun}(\alpha), f)$;
- $\mathsf{get\_fun}(\mathsf{Vis}_i(x, k), f) = \mathsf{Vis}_i(x, \blacktriangleright\mathsf{get\_fun}(-, f) \circ k)$.

In the next section we will see how to use get_fun to define an application function $\alpha \bullet \beta$ for applying a GITree function α to a GITree argument β.

In the rest of the paper we will define other operations on GITrees using just the computational rules, with the understanding that we can write down the explicit recursor for any such set of equations. Interested readers are referred to the Coq formalization for the details.

3.2 Programming with GITrees

Using the recursion principle we can define operations on GITrees that correspond to common programming constructs. For example, with get_fun we can write a function $\mathsf{App}_l(\alpha, \beta)$, which applies α to β if α is a function, and returns $\mathsf{Err}(\mathit{RunTime})$ otherwise.

$$\mathsf{App}_l(\alpha, \beta) = \mathsf{get\_fun}(\alpha, \lambda g.\, \mathsf{Tau}(\blacktriangleright g(\beta))).$$

This operation gives us "call-by-name" application, in the sense that it satisfies

$$\mathsf{App}_l(\mathsf{Fun}(\mathsf{next}(g)), \beta) = \mathsf{Tick}(g(\beta))$$

for $g : \mathbf{IT}_E(A) \to \mathbf{IT}_E(A)$ for any argument β. In particular, it invokes the underlying function g even if the argument β is a Tick or an effect, without evaluating the argument first. In order to define a "call-by-value" application, we compose App_l with the following operation.

The function $\mathsf{get\_val}(\alpha, f)$ recurses into its argument, looking under Ticks and Vis's, until it reaches either a function or a ground type (i.e. a value from $\mathbf{IT}^v_E(A)$), after which it applies the function $f : \mathbf{IT}_E(A) \to \mathbf{IT}_E(A)$ to it:

$$\begin{aligned}
\mathsf{get\_val}(\mathsf{Ret}(a), f) &= f(\mathsf{Ret}(a)) & \mathsf{get\_val}(\mathsf{Fun}(g), f) &= f(\mathsf{Fun}(g)) \\
\mathsf{get\_val}(\mathsf{Err}(e), f) &= \mathsf{Err}(e) & \mathsf{get\_val}(\mathsf{Tick}(\alpha), f) &= \mathsf{Tick}(\mathsf{get\_val}(\alpha, f)) \\
\mathsf{get\_val}(\mathsf{Vis}_i(x, k), f) &= \mathsf{Vis}_i(x, \blacktriangleright\mathsf{get\_val}(-, f) \circ k)
\end{aligned}$$

As syntactic sugar, we write $\mathsf{Let}\, x = \alpha\, \mathsf{in}\, \beta(x)$ for $\mathsf{get\_val}(\alpha, \lambda x.\, \beta(x))$.

Now we can define the "call-by-value" application $\alpha \bullet \beta$ as $\mathsf{get\_val}\left(\beta, \lambda \beta_v.\, \mathsf{App}_l(\alpha, \beta_v)\right)$. This strict application then satisfies the following computational rules

$$\begin{aligned}
\alpha \bullet \mathsf{Tick}(\beta) &= \mathsf{Tick}(\alpha \bullet \beta) & \alpha \bullet \mathsf{Vis}_i(x, k) &= \mathsf{Vis}_i(x, \lambda y.\, \mathsf{next}(\alpha)\ (\blacktriangleright\bullet)\ k\ y) \\
\mathsf{Tick}(\alpha) \bullet \beta_v &= \mathsf{Tick}(\alpha \bullet \beta_v) & \mathsf{Vis}_i(x, k) \bullet \beta_v &= \mathsf{Vis}_i(x, \lambda y.\, k\ y\ (\blacktriangleright\bullet)\ \mathsf{next}(\beta_v)) \\
\mathsf{Fun}(\mathsf{next}(g)) \bullet \beta_v &= \mathsf{Tick}(g(\beta_v)) & \alpha \bullet \beta &= \mathsf{Err}(\mathit{RunTime}) \text{ in other cases}
\end{aligned}$$

Where $- (\blacktriangleright\bullet) -$ is the lifting of $- \bullet -$ to $\blacktriangleright\mathbf{IT}_E(A) \to \blacktriangleright\mathbf{IT}_E(A) \to \blacktriangleright\mathbf{IT}_E(A)$, and $\beta_v \in \mathbf{IT}^v_E(A)$ is either $\mathsf{Ret}(a)$ or $\mathsf{Fun}(g)$. The application function $- \bullet -$ not only simulates strict application, but it also fixes a right-to-left evaluation order of effects and computation steps.

One can see that there are common properties for the computational rules between $\mathsf{get\_fun}(-, f)$, $\mathsf{App}_l(-, \beta)$, $\alpha \bullet -$, and $- \bullet \beta_v$ (where $\beta_v \in \mathbf{IT}^v_E(A)$): they all preserve ticks, effects, and errors. Functions that have these preservation properties are called *homomorphisms* of GITrees and will play an important role in later sections.

Definition 3.4. A function $f : \mathbf{IT}_E(A) \to \mathbf{IT}_E(A)$ is a *homomorphism*, written as $f \in \mathit{Hom}$, if it satisfies the following equations:

- $f(\mathsf{Err}(e)) = \mathsf{Err}(e)$;
- $f(\mathsf{Tick}(\alpha)) = \mathsf{Tick}(f(\alpha))$;
- $f(\mathsf{Vis}_i(x, k)) = \mathsf{Vis}_i(x, \blacktriangleright f \circ k)$

As expected from the name, the identity function is a homomorphism and composition of two homomorphisms is a homomorphism. This notion of homomorphism is inspired by the one in [Hoshino 2012]. It follows from the definition, that in order to define a homomorphism it suffices to define its action on GITree values.

Programming with GITrees and natural numbers. In the remainder of this paper we work with a lot of examples involving programming with natural numbers (as an illustrative ground type). It is then useful to assume in the remainder of this paper that the ground type A in any type $\mathbf{IT}_E(A)$ of GITrees is "large enough" to contain natural numbers, and the unit type. That is, we assume that $A \simeq \mathbf{1} + \mathsf{Nat} + \ldots$, and we simply write $\mathsf{Ret}(n)$ for $\mathsf{Ret}(\mathsf{inj}(n))$ and $\mathsf{Ret}(())$ for $\mathsf{Ret}(\mathsf{inj}'(()))$ (for appropriate injections $\mathsf{inj} : \mathsf{Nat} \to A$ and $\mathsf{inj}' : \mathbf{1} \to A$). We will also abbreviate $\mathbf{IT}_E(A)$ as $\mathbf{IT}$ or $\mathbf{IT}_E$ when A is generic as above or is clear from the context.

In Figure 3 we summarize the operations on GITrees that we define using recursion and other functions. The computational rules described in Figure 3 are only for the base cases; the other computational rules follow from the fact that those operations are homomorphisms. Concretely, we

$$\frac{n \in \mathsf{Nat}}{\mathsf{get\_nat}(\mathsf{Ret}(n), f) = f(n)} \qquad \frac{b \notin \mathsf{Nat}}{\mathsf{get\_nat}(\mathsf{Ret}(b), f) = \mathsf{Err}(\mathit{RunTime})}$$

$$\mathsf{get\_nat}(\mathsf{Fun}(g), f) = \mathsf{Err}(\mathit{RunTime}) \qquad \mathsf{get\_fun}(\mathsf{Ret}(a), f) = \mathsf{Err}(\mathit{RunTime}) \qquad \mathsf{get\_fun}(\mathsf{Fun}(g), f) = f(g)$$

$$\mathsf{If}(\mathsf{Ret}(0), \alpha_1, \alpha_2) = \alpha_1 \qquad \frac{n > 0}{\mathsf{If}(\mathsf{Ret}(n), \alpha_1, \alpha_2) = \alpha_2} \qquad \mathsf{If}(\mathsf{Fun}(f), \alpha_1, \alpha_2) = \mathsf{Err}(\mathit{RunTime})$$

$$\frac{b \notin \mathsf{Nat}}{\mathsf{If}(\mathsf{Ret}(b), \alpha_1, \alpha_2) = \mathsf{Err}(\mathit{RunTime})} \qquad \frac{n_1, n_2 \in \mathsf{Nat}}{\mathsf{NatOp}_f(\mathsf{Ret}(n_1), \mathsf{Ret}(n_2)) = \mathsf{Ret}(f(n_1, n_2))}$$

$$\frac{\alpha_v \text{ or } \beta_v \text{ are not } \mathsf{Ret}(n)}{\mathsf{NatOp}_f(\alpha_v, \beta_v) = \mathsf{Err}(\mathit{RunTime})} \qquad \beta_v \;;\; \alpha = \alpha \qquad \mathsf{While}\,\alpha\,\mathsf{do}\,\beta = \mathsf{If}\big(\alpha, (\beta \;;\; \mathsf{Tick}(\mathsf{While}\,\alpha\,\mathsf{do}\,\beta)), \mathsf{Ret}(())\big)$$

Fig. 3. Programming operations on GITrees.

have the following operations. The get_nat function extracts a natural number from a GITree and applies the function $f : \mathsf{Nat} \to \mathbf{IT}$ to it. It is a homomorphism in the first argument. If it encounters a function $\mathsf{Fun}(g)$ or a different ground value $\mathsf{Ret}(b)$, then it returns an error. The If operations test whether the first argument is zero, and picks the appropriate branch. The function $\mathsf{If}(-, \alpha_1, \alpha_2)$ is a homomorphism. Similarly, if the first argument is not a natural number then If returns an error. The NatOp_f operation applies the binary function f to its integer arguments, returning an error on all the other values. The maps $\mathsf{NatOp}_f(\alpha, -)$ and $\mathsf{NatOp}_f(-, \beta_v)$ are homomorphisms for $\beta_v \in \mathbf{IT}^v$. The $\alpha \;;\; \beta$ is a sequencing operation: it puts all the effects and ticks in α before the effects and ticks in β. This is witnessed by the fact that $(-) \;;\; \alpha$ is a homomorphism. The $\mathsf{While}\,\alpha\,\mathsf{do}\,\beta$ represents a while loop with the conditional α and the body β; it is defined using guarded recursion, and is equal to its one-step unfolding using the If construct.

Let us look at some example programs that we can write using the operations we have defined.

Example 3.5 (Factorial). In the first example, we have a factorial function that we implement using the store operations (Example 3.2).

$$\begin{aligned}
\mathsf{fact}(n) &\triangleq \mathsf{Alloc}\,(\mathsf{Ret}(1), \lambda acc.\, \mathsf{Alloc}(\mathsf{Ret}(n), \lambda \ell.\, \mathsf{factBody}(acc, \ell) \;;\; \mathsf{Read}(acc)))\\
\mathsf{factBody}(acc, \ell) &\triangleq \mathsf{While}\,\mathsf{Read}(\ell)\,\mathsf{do}\\
&\qquad \mathsf{Let}\, i = \mathsf{Read}(\ell)\,\mathsf{in}\\
&\qquad \mathsf{Let}\, r = \mathsf{NatOp}_\times(i, \mathsf{Read}(acc))\,\mathsf{in}\\
&\qquad \mathsf{Let}\, i = \mathsf{NatOp}_-(i, \mathsf{Ret}(1))\,\mathsf{in}\\
&\qquad \mathsf{Write}(acc, r) \;;\; \mathsf{Write}(\ell, i)
\end{aligned}$$

The program factBody computes the factorial of the number stored in the location ℓ using an intermediate location *acc* for the accumulated result. The complete program fact then allocates the needed references and runs factBody before reading off the result from the location *acc*.

Example 3.6 (Encoding of pairs). Our definition of GITrees does not include arbitrary algebraic datatypes, like pairs or sums. We can, however, encode them using a Church-style encoding. We write $(\alpha, \beta) : \mathbf{IT}$ for the guarded interaction tree

$$\mathsf{Let}\, y = \beta\,\mathsf{in}\,\mathsf{Let}\, x = \alpha\,\mathsf{in}\,\mathsf{Fun}(\mathsf{next}(\lambda f.\, f \bullet x \bullet y)).$$

Note that (α_v, β_v) is a GITree value whenever α_v and β_v are. Furthermore, $(\alpha, -)$ and $(-, \beta_v)$ are homomorphisms. We then define the projection functions as

$$\pi_1\,(\alpha) = \alpha \bullet \mathsf{Fun}(\mathsf{next}(\lambda a.\, \mathsf{Fun}(\mathsf{next}(\lambda b.\, a)))) \qquad \pi_2\,(\alpha) = \alpha \bullet \mathsf{Fun}(\mathsf{next}(\lambda a.\, \mathsf{Fun}(\mathsf{next}(\lambda b.\, b)))).$$

The projection functions then satisfy the following computational rules:

$$\pi_1\,(\alpha_v, \beta_v) = \mathsf{Tick}^3(\alpha_v) \qquad \pi_2\,(\alpha_v, \beta_v) = \mathsf{Tick}^3(\beta_v).$$

We can use similar style encodings to represent other algebraic datatypes as guarded interaction trees.

4 REIFICATION OF EFFECTS AND REDUCTIONS OF GITREES

GITrees allow us to conveniently write down and combine various effects. But in order to reason about the effects we also need a way of giving them meaning. In this section we establish a way of *reifying* effects of GITrees and use reification to define *reductions* of GITrees, which explain how computations represented by GITrees reduce.

In order to interpret stateful effects we assume that we have a type *State*, and each effect is interpreted using the state monad with a function:

$$r : \prod_{\mathtt{i} \in E} \mathit{Ins}_{\mathtt{i}}(\blacktriangleright \mathbf{IT}_E) \times \mathit{State} \longrightarrow \mathit{option}(\mathit{Outs}_{\mathtt{i}}(\blacktriangleright \mathbf{IT}_E) \times \mathit{State}).$$

We call a tuple (E, State, r) a *reifier* for the effects E. Assuming we have such a reifier, we write a function reify : $\mathbf{IT} \times \mathit{State} \longrightarrow \mathbf{IT} \times \mathit{State}$ that satisfies

$$\frac{r_i(x, \sigma) = \mathsf{Some}(y, \sigma') \qquad k\; y = \mathsf{next}(\beta)}{\mathsf{reify}(\mathsf{Vis}_i(x, k), \sigma) = (\mathsf{Tick}(\beta), \sigma')} \qquad \frac{r_i(x, \sigma) = \mathsf{None}}{\mathsf{reify}(\mathsf{Vis}_i(x, k), \sigma) = (\mathsf{Err}(\mathit{RunTime}), \sigma)}$$

Example 4.1 (Reification for the input/output operations Example 3.1). We take the state *State* to be a pair of two lists of natural numbers, corresponding to input and output tapes. The reifier is defined as

$$r_{\mathtt{input}}((), (n\vec{n}, \vec{m})) = \mathsf{Some}(n, (\vec{n}, \vec{m})) \qquad r_{\mathtt{input}}((), (\epsilon, \vec{m})) = \mathsf{None}$$
$$r_{\mathtt{output}}(x, (\vec{n}, \vec{m})) = \mathsf{Some}((), (\vec{n}, x\vec{m}))$$

Example 4.2 (Reification for the higher-order store operations Example 3.2). For higher-order store we take *State* to be the type of finite partial maps $\mathit{Loc} \xrightarrow{\text{fin}} \blacktriangleright \mathbf{IT}$. The reifier function is defined in the expected way:

$$r_{\mathtt{alloc}}(\alpha, \sigma) = \mathsf{Some}(\ell, \sigma[\ell \mapsto \alpha]) \qquad \text{where } \ell \text{ is the smallest location not present in } \sigma$$

$$r_{\mathtt{read}}(\ell, \sigma) = \begin{cases} \mathsf{Some}(\alpha, \sigma) & \text{if } \sigma(\ell) = \alpha \\ \mathsf{None} & \text{otherwise} \end{cases}$$

$$r_{\mathtt{write}}((\ell, \beta), \sigma) = \begin{cases} \mathsf{Some}((), \sigma[\ell \mapsto \beta]) & \text{if } \sigma(\ell) \text{ is defined} \\ \mathsf{None} & \text{otherwise} \end{cases}$$

$$r_{\mathtt{dealloc}}(\ell, \sigma) = \begin{cases} \mathsf{Some}((), \sigma \setminus \{\ell\}) & \text{if } \sigma(\ell) \text{ is defined} \\ \mathsf{None} & \text{otherwise} \end{cases}$$

From reification to reductions. Using the reification function we can formulate the reduction relation on interaction trees. The *internal* reduction relation $\rightsquigarrow$: $(\mathbf{IT}\times \mathit{State}) \to (\mathbf{IT}\times \mathit{State}) \to \mathrm{iProp}$:

$$(\alpha, \sigma) \rightsquigarrow (\beta, \sigma') \triangleq \left(\alpha = \mathsf{Tick}(\beta) \land \sigma = \sigma'\right) \lor \left(\exists i\, x\, k.\, \alpha = \mathsf{Vis}_i(x, k) \land \mathsf{reify}(\alpha, \sigma) = (\mathsf{Tick}(\beta), \sigma')\right)$$

Intuitively, a reduction of GITrees corresponds to either stripping away one computational step, or to reifying an effect. We consider an (annotated) transitive closure of the reduction relation:

$$(\alpha, \sigma) \rightsquigarrow^0 (\beta, \sigma') \triangleq \alpha = \beta \land \sigma = \sigma'$$
$$(\alpha, \sigma) \rightsquigarrow^{n+1} (\beta, \sigma') \triangleq \exists \alpha_0, \sigma_0.\, (\alpha, \sigma) \rightsquigarrow (\alpha_0, \sigma_0) \land (\alpha_0, \sigma_0) \rightsquigarrow^n (\beta, \sigma')$$

We write $\rightsquigarrow^*$ for the reflexive transitive closure of the reduction relation.

Reductions and homomorphisms. Homomorphisms (Definition 3.4) play an important role in the reduction relation, allowing us to compute the reductions more easily. Specifically, homomorphisms preserve and reflect reductions:

Lemma 4.3. *Let f be a homomorphism. Then $(\alpha, \sigma) \rightsquigarrow (\beta, \sigma')$ implies $(f(\alpha), \sigma) \rightsquigarrow (f(\beta), \sigma')$.*

Lemma 4.4. *Let f be a homomorphism. If $(f(\alpha), \sigma) \rightsquigarrow (\beta', \sigma')$ then either*

- *α is a GITree-value, or;*
- *there exists β such that $(\alpha, \sigma) \rightsquigarrow (\beta, \sigma')$ and $\triangleright(f(\beta) = \beta')$.*

These two lemmas suggest that homomorphisms play the role of evaluation contexts within the reduction relation $\rightsquigarrow$. For example, if $(\alpha \ ; \ \beta, \sigma) \rightsquigarrow (\delta, \sigma')$, then either α was a value, or $(\alpha, \sigma) \rightsquigarrow (\alpha', \sigma')$ and $\triangleright(\alpha' \ ; \beta = \delta)$.

Continuation-independent reifiers. The reifiers that we consider here produce an output based on the input, but do not have direct access to the continuation. The reification function reify just calls the continuation with the produced output. This *continuation-independence* is crucial for proving Lemma 4.4 (and the associated rule wp-hom in separation logic in Section 6). Not all effects are continuation-independent, for example `call/cc` cannot be implemented this way. In this paper, just like in [Xia et al. 2019], we stick to working with continuation-independent effects, as it simplifies the separation logic and the reasoning principles, and we defer studying continuation-dependent effects to future work.

5 MODELING A HIGHER-ORDER EFFECTFUL PROGRAMMING LANGUAGE

In this section we show how guarded interaction trees provide a model for a programming language with recursion, higher-order functions, and effects. Specifically, we study a PCF-like higher-order programming language with input/output effects, give its interpretation into $\mathbf{IT}_{io}$ (see Examples 3.1 and 4.1), and show its soundness, i.e., that the interpretation agrees with the operational semantics. The same approach applies to other classes of effects for which you can write operational semantics.

Syntax and operational semantics. The syntax for the programming language, which we dub $\lambda_{\mathrm{rec,io}}$, consists of values and expressions:

$$v \in \mathit{Val} ::= n \mid \mathsf{rec}\, f(x) = e$$
$$e \in \mathit{Expr} ::= x \mid v \mid \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \mid e_1(e_2) \mid e_1 + e_2 \mid e_1 - e_2 \mid \mathsf{input} \mid \mathsf{output}(e)$$

where n ranges over the set of natural numbers, and f, x range over the set *Var* of variables.

The operational semantics for $\lambda_{\mathrm{rec,io}}$ is given in Figure 4 as a small-step reduction relation on the configurations *Expr* × *State*, where *State* is a pair of lists as in Example 4.1. The reductions are

$$\textsc{red-beta}\quad ((\mathsf{rec}\, f(x) = e)\ v, \sigma) \longrightarrow (e[v/x][\mathsf{rec}\, f(x) = e/f], \sigma)$$

$$\textsc{red-natop}\quad \frac{n_1, n_2 \in \mathbb{N} \qquad \oplus \in \{+, -, \times, \dots\} \qquad n_1 \oplus n_2 = n}{(n_1 \oplus n_2, \sigma) \longrightarrow (n, \sigma)}$$

$$\textsc{red-if-false}\quad (\mathsf{if}\ 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, \sigma) \longrightarrow (e_2, \sigma)$$

$$\textsc{red-if-true}\quad \frac{n \in \mathbb{N} \qquad n > 0}{(\mathsf{if}\ n\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, \sigma) \longrightarrow (e_1, \sigma)}$$

$$\textsc{red-input}\quad (\mathsf{input}, (n'\vec{n}, \vec{m})) \longrightarrow (n', (\vec{n}, \vec{m}))$$

$$\textsc{red-output}\quad (\mathsf{output}(m), (\vec{n}, \vec{m})) \longrightarrow (0, (\vec{n}, m'\vec{m}))$$

$$\textsc{red-ectx}\quad \frac{(e_1, \sigma_1) \longrightarrow (e_2, \sigma_2)}{(K[e_1], \sigma_1) \longrightarrow (K[e_1], \sigma_2)}$$

Fig. 4. Small-step operational semantics for $\lambda_{\mathrm{rec,io}}$.

$$[\![x]\!]_\rho = \rho(x) \qquad [\![n]\!]_\rho = \mathsf{Ret}(n) \qquad [\![\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]_\rho = \mathsf{If}([\![e]\!]_\rho, [\![e_1]\!]_\rho, [\![e_2]\!]_\rho)$$

$$\frac{\oplus \in \{+, -, \times, \dots\}}{[\![e_1 \oplus e_2]\!]_\rho = \mathsf{NatOp}_\oplus([\![e_1]\!]_\rho, [\![e_2]\!]_\rho)} \qquad [\![\mathsf{input}]\!]_\rho = \mathsf{Input} \qquad [\![\mathsf{output}(e)]\!]_\rho = \mathsf{get\_nat}([\![e]\!]_\rho, \mathsf{Output})$$

$$[\![e_1\ e_2]\!]_\rho = [\![e_1]\!]_\rho \bullet [\![e_2]\!]_\rho \qquad [\![\mathsf{rec}\, f(x) = e]\!]_\rho = \mathit{fix}_{\mathrm{IT}}(\lambda(t : \blacktriangleright\mathbf{IT}).\, \mathsf{Fun}(\blacktriangleright(\lambda\alpha\ v.\, [\![e]\!]_{\rho[x\mapsto v][f\mapsto\alpha]})(t))).$$

Fig. 5. Semantic interpretation for $\lambda_{\mathrm{rec,io}}$.

defined, following [Felleisen and Hieb 1992], using *evaluation contexts* $K \in \mathit{Ectx}$, given as:

$$K \in \mathit{Ectx} ::= [\,\bullet\,] \mid \mathsf{output}(K) \mid \mathsf{if}\ K\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid e\ K \mid K\ v \mid e \oplus K \mid K \oplus v$$

By $K[e]$ we denote the result of replacing the hole $[\,\bullet\,]$ in the context K with the expression e. The evaluation contexts ensure the call-by-value right-to-left evaluation order of $\lambda_{\mathrm{rec,io}}$, as having a predefined evaluation order is important in the presence of effects.

Interpretation in guarded interaction trees. We will interpret a closed program e as an interaction tree $[\![e]\!] : \mathbf{IT}_{io}(A)$. The effects io are those of Examples 3.1 and 4.1, and we assume that the ground type A is "large enough" to have natural numbers. For convenience, we drop the ground type and write simply $\mathbf{IT}_{io}$ for $\mathbf{IT}_{io}(A)$.

In order to provide a (compositional) denotational semantics we need to provide an interpretation not only for closed terms, but for open terms as well. Given a set $\mathrm{fv}(e) = \{x_1, \dots, x_n\}$ of free variables of e, we define the interpretation $[\![e]\!]_\rho : \mathbf{IT}_{io}$, where ρ maps the free variables of e to interaction trees. The interpretation function is defined in Figure 5. The definition follows the standard notion of semantics for (untyped) λ-calculus, adjusted for effects and explicit recursion. The interpretation of recursive functions $\mathsf{rec}\, f(x) = e$ is defined using the guarded fixed pointed operation $\mathit{fix}_{\mathrm{IT}} : (\blacktriangleright\mathbf{IT} \to \mathbf{IT}) \to \mathbf{IT}$, and satisfies the following equality:

$$[\![\mathsf{rec}\, f(x) = e]\!]_\rho = \mathsf{Fun}(\mathsf{next}(\lambda v.\, [\![e]\!]_{\rho[x\mapsto v][f\mapsto[\![\mathsf{rec}\, f(x)=e]\!]_\rho]}))$$

We show that the interpretation is sound:

Theorem 5.1 (Soundness). *If* $(e_1, \sigma_1) \longrightarrow (e_2, \sigma_2)$, *then* $([\![e_1]\!], \sigma_1) \rightsquigarrow^* ([\![e_2]\!], \sigma_2)$.

We prove Theorem 5.1 by induction on the $\longrightarrow$-derivation. The most interesting cases are for the reductions red-beta and red-ectx, which we now sketch. For the former, we need a substitution lemma, and for the latter we need to extend the interpretation to evaluation contexts.

$$\textsc{wp-val}\ \frac{\alpha \in \mathbf{IT}^v \qquad \Phi(\alpha)}{\mathsf{wp}\ \alpha\ \{\Phi\}} \qquad \textsc{wp-tick}\ \frac{\triangleright \mathsf{wp}\ \alpha\ \{\Phi\}}{\mathsf{wp}\ \mathsf{Tick}(\alpha)\ \{\Phi\}} \qquad \textsc{wp-hom}\ \frac{f \in \mathit{Hom} \qquad \mathsf{wp}\ \alpha\ \{\beta_v.\, \mathsf{wp}\ f(\beta_v)\ \{\Phi\}\}}{\mathsf{wp}\ f(\alpha)\ \{\Phi\}}$$

$$\textsc{wp-reify}\ \frac{\mathsf{has\_state}(\sigma) \qquad \mathsf{reify}(\mathsf{Vis}_i(x,k), \sigma) = (\mathsf{Tick}(\beta), \sigma') \qquad \triangleright(\mathsf{has\_state}(\sigma') \mathrel{-\!\!*} \mathsf{wp}\ \beta\ \{\Phi\})}{\mathsf{wp}\ \mathsf{Vis}_i(x,k)\ \{\Phi\}}$$

$$\textsc{wp-upd}\ \frac{\Rrightarrow \mathsf{wp}\ \alpha\ \{\alpha_v.\ \Rrightarrow \Phi(\alpha_v)\}}{\mathsf{wp}\ \alpha\ \{\Phi\}} \qquad \textsc{wp-mono}\ \frac{\mathsf{wp}\ \alpha\ \{\Psi\} \qquad \forall \alpha_v.\, \Psi(\alpha_v) \mathrel{-\!\!*} \Phi(\alpha_v)}{\mathsf{wp}\ \alpha\ \{\Phi\}} \qquad \textsc{wp-lam}\ \frac{\mathsf{wp}\ \beta\ \{\beta_v.\, \triangleright \mathsf{wp}\ f(\beta_v)\ \{\Phi\}\}}{\mathsf{wp}\ \mathsf{Fun}(\mathsf{next}(f)) \bullet \beta\ \{\Phi\}}$$

Fig. 6. Selected weakest precondition calculus rules.

Lemma 5.2 (Substitution lemma). *For any expression e with a free variable x we have*

$$[\![e[e'/x]]\!]_\rho = [\![e]\!]_{\rho[x \mapsto [\![e']\!]]}.$$

Proof. By induction on e, using Löb induction in the case of recursive functions. □

In order to handle red-ectx we provide the following auxiliary interpretation for evaluation contexts. Each evaluation context K is interpreted as a *homomorphism* $[\![K]\!]_\rho : \mathbf{IT} \to \mathbf{IT}$, such that $[\![K[e]]\!]_\rho = [\![K]\!]_\rho([\![e]\!]_\rho)$, which together with Lemma 4.3 implies the soundness of the red-ectx reduction.

6 SEPARATION LOGIC OVER GITREES

In this section we define a separation logic as a program logic for guarded interaction trees. We define a proposition $\mathsf{wp}\ \alpha\ \{\Phi\}$ to denote that an interaction tree α is safe to reduce, and if α reduces to an interaction tree value β_v, then β_v satisfies the postcondition $\Phi : \mathbf{IT}^v \to \mathrm{iProp}$.

In this section we make use of the separation logic connectives of Iris, which we recall here:

$$P ::= \ \ldots \mid P * P \mid P \mathrel{-\!\!*} P \mid \ \Rrightarrow P \mid \Box P \mid \boxed{P} \ \mid \ \ldots$$

For brevity, we only briefly recall the intuitive reading of these propositions and refer to [Jung et al. 2018] for details. The proposition $P * Q$ says that the propositions P and Q hold over disjoint resources; the proposition $P \mathrel{-\!\!*} Q$ says that if we were to add any resources which satisfy P, then Q would be satisfied. The proposition $\Rrightarrow P$ says that the current resources can be updated to satisfy P. The proposition $\Box P$ states that P holds *persistently*, i.e., without asserting any resources. Crucially, such propositions can be duplicated: $\Box P \vdash \Box P * \Box P$. An example of a persistent proposition is the invariant proposition $\boxed{P}$, which satisfies $\boxed{P} \vdash \Box \boxed{P}$.

6.1 Weakest Precondition Rules

Selected rules for the weakest precondition proposition $\mathsf{wp}\ \alpha\ \{\Phi\}$ are given in Figure 6. The rule wp-val states that to verify a value it suffices to check that the value satisfies the postcondition. The rule wp-tick states that in order to verify $\mathsf{Tick}(\alpha)$ it suffices to verify α, under a later $\triangleright$. The rule wp-hom states that in order to verify $f(\alpha)$ for a homomorphism f, it suffices to reduce α to a value α_v, and then verify $f(\alpha_v)$.

The rule wp-reify tells us how to deal with effects. The rule uses the proposition $\mathsf{has\_state}(\sigma)$ which signifies the *exclusive ownership* of the current state σ. The use of separation logic is crucial in this case, as we do not want to allow duplicating that proposition. The rule then states that in order to verify an effect, one has to provide the current state σ' and the proof that the interaction tree with the effect reifies into some $\mathsf{Tick}(\beta)$. Then, the user has to verify that the resulting β reduces to a value satisfying the postcondition, under the assumption that the state has been updated to σ'.

The rule wp-upd states that one can update ghost resources before and after reducing α. The rule wp-mono states that one can always weaken the postcondition in $\mathrm{wp}\ \alpha\ \{\Phi\}$. Finally, wp-lam is an example of a derived rule. It combines the computational rule for function application of GITrees, and rules wp-hom and wp-tick. Let us look at an example derivation using these rules.

Example 6.1. Consider a $\lambda_{\mathrm{rec,io}}$ expression ($\mathsf{input} + 1$). It is interpreted as the GITree $[\![\mathsf{input} + 1]\!] = \mathsf{NatOp}_+(\mathsf{Input}, \mathsf{Ret}(1))$, for which we can prove the following specification:

$$\frac{\mathsf{has\_state}(n\vec{n}, \vec{m}) \qquad \triangleright(\mathsf{has\_state}(\vec{n}, \vec{m}) \twoheadrightarrow \Phi(\mathsf{Ret}(n+1)))}{\mathsf{wp}\ \mathsf{NatOp}_+(\mathsf{Input}, \mathsf{Ret}(1))\ \{\Phi\}}$$

Proof. Note that $\mathsf{NatOp}_+(-, \mathsf{Ret}(1))$ is a homomorphism. We apply wp-hom, reducing our goal to:

$$\mathsf{wp}\ \mathsf{Input}\ \{\beta_v.\ \mathsf{wp}\ \mathsf{NatOp}_+(\beta_v, \mathsf{Ret}(1))\ \{\Phi\}\}.$$

At this point we can use the assumption $\mathsf{has\_state}(n\vec{n}, \vec{m})$ and the rule wp-reify. By the reifier of input/output effects, $\mathsf{reify}(\mathsf{Input}, (n\vec{n}, \vec{m})) = \mathsf{Some}(\mathsf{Tick}(\mathsf{Ret}(n)), (\vec{n}, \vec{m}))$, and we get the following goal:

$$\triangleright(\mathsf{has\_state}(\vec{n}, \vec{m}) \twoheadrightarrow \mathsf{wp}\ \mathsf{Ret}(n)\ \{\beta_v.\ \mathsf{wp}\ \mathsf{NatOp}_+(\beta_v, \mathsf{Ret}(1))\ \{\Phi\}\}).$$

Recall that we still have the assumption $\triangleright(\mathsf{has\_state}(\vec{n}, \vec{m}) \twoheadrightarrow \Phi(\mathsf{Ret}(n+1)))$. By the monotonicity of $\triangleright$ we can remove the $\triangleright$ modality both from the goal and the assumption. Since $\mathsf{Ret}(n)$ is a GITree value, we can use wp-val and reduce the goal to

$$\mathsf{wp}\ \mathsf{NatOp}_+(\mathsf{Ret}(n), \mathsf{Ret}(1))\ \{\Phi\}.$$

By calculation, $\mathsf{NatOp}_+(\mathsf{Ret}(n), \mathsf{Ret}(1)) = \mathsf{Ret}(n+1)$, which is also a GITree value. We can then apply wp-val again to reduce the goal to $\Phi(\mathsf{Ret}(n+1))$, which follows from the assumption. □

We define the weakest precondition as a guarded recursive predicate, as is standard in Iris. The weakest precondition then satisfies the following *adequacy* and *safety* theorem, the proof of which relies on the adequacy of Iris (Theorem 2.1).

Theorem 6.2. *Let α be an interaction tree and σ be a state such that*

$$\mathsf{has\_state}(\sigma) \vdash wp\ \alpha\ \{\Phi\}$$

is derivable for some meta-level predicate Φ (containing only intuitionistic logic connectives). Then for any β and σ' such that $(\alpha, \sigma) \rightsquigarrow^* (\beta, \sigma')$, one of the following two things hold:

- (adequacy) *either $\beta \in \mathbf{IT}^v$, and $\Phi(\beta)$ holds in the meta-logic;*
- (safety) *or there are β_1 and σ_1 such that $(\beta, \sigma') \rightsquigarrow (\beta_1, \sigma_1)$*

In particular, safety implies that $\beta \neq \mathtt{Err}(e)$ for any error $e \in \mathtt{Error}$.[3]

Finally, it is worth noting that separation logic/Iris is useful for reasoning about higher-order GITrees even in the absence of effects, as demonstrated by the following example.

Example 6.3. Using guarded recursion, we can write down a GITree Iter that satisfies the equation:

$$\mathsf{Iter} \bullet f \bullet \alpha \bullet \beta = \mathsf{If}(\alpha, f \bullet (\mathsf{Iter} \bullet f \bullet \mathsf{NatOp}_-(\alpha, \mathsf{Ret}(1)) \bullet \beta), \beta).$$

[3]While the weakest precondition that we presented in this section disallow any errors in guarded interaction tree, we will consider in Section 9.3 that allows *some* errors, at the user's discretion.

That is, Iter $\bullet\ f\ \bullet$ Ret$(n)\ \bullet\ \beta$ computes the iterated application $f^n\ \bullet\ \beta$. We can give Iter the following higher-order specification:

$$\frac{\mathsf{wp}\ \beta\ \{\Psi\} \qquad \Box\forall\beta_v.\ \Psi(\beta_v) \twoheadrightarrow \mathsf{wp}\ f \bullet \beta_v\ \{\Psi\}}{\mathsf{wp}\ (\mathsf{Iter} \bullet f \bullet \mathsf{Ret}(n) \bullet \beta)\ \{\Psi\}}$$

The specification says that if β initially satisfies Ψ and f preserves Ψ , then Iter $\bullet\ f\ \bullet$ Ret$(n)\ \bullet\ \beta$ will also satisfy Ψ. The second premise is the specification of f, and it can be used multiple times in the proof. For that reason the that premise is behind the persistently modality $\Box$.

It is also worth noting that while Iter itself does not use state, the function f that we supply to it might as well use all sorts of effects internally, and our implementation and specification of Iter is oblivious to that.

6.2 Domain-Specific Logic for Higher-Order Store

Now we show how we can use the standard mechanisms in Iris to recover a fairly standard-looking separation logic for a programming language with references from the weakest precondition calculus presented above. We use Iris's notion of *higher-order ghost state* [Jung et al. 2016, 2018] to provide the following logical interface for the higher-order store operations:

WP-ALLOC
$$\frac{\mathsf{heap\_ctx} \qquad \triangleright\forall\ell.\ \ell \mapsto \alpha \twoheadrightarrow \mathsf{wp}\ k\ \ell\ \{\Phi\}}{\mathsf{wp}\ \mathsf{Alloc}(\alpha, k)\ \{\Phi\}}$$

WP-READ
$$\frac{\mathsf{heap\_ctx} \qquad \triangleright\ell \mapsto \alpha \qquad \triangleright(\ell \mapsto \alpha \twoheadrightarrow \mathsf{wp}\ \alpha\ \{\Phi\})}{\mathsf{wp}\ \mathsf{Read}(\ell)\ \{\Phi\}}$$

WP-WRITE
$$\frac{\mathsf{heap\_ctx} \qquad \triangleright\ell \mapsto \alpha \qquad \triangleright(\ell \mapsto \beta \twoheadrightarrow \Phi(\mathsf{Ret}(())))}{\mathsf{wp}\ \mathsf{Write}(\ell, \beta)\ \{\Phi\}}$$

WP-DEALLOC
$$\frac{\mathsf{heap\_ctx} \qquad \triangleright\ell \mapsto \alpha \qquad \triangleright\Phi(\mathsf{Ret}(()))}{\mathsf{wp}\ \mathsf{Dealloc}(\ell)\ \{\Phi\}}$$

$$\mathsf{heap\_ctx} \vdash \Box\mathsf{heap\_ctx}$$

Here heap_ctx is a persistent proposition, which is part of the logical interface.

Thus our goal is to provide definitions of heap_ctx and $\ell \mapsto \alpha$ that allow us to derive the rules above. The main challenge is that the resource has_state(σ) provides a singular complete view of the state, without the ability to split it into local portions corresponding to individual locations. That has_state is not splittable by itself is not surprising – it is an abstract representation of an arbitrary state for arbitrary effects, and there is no a priori way of splitting it. However, for our specific state (a heap $Loc \xrightarrow{\text{fin}} \blacktriangleright\mathbf{IT}$) we know how to do the splitting. What we need to do is to provide an alternative view of the state, amenable to splitting, and tie it together with the actual state of the has_state predicate.

Our first step is then to provide a *resource algebra* for this view of the state. Following the standard practice of Iris, we use an authoritative resource algebra of the heap. It contains two kinds of resources: the "full heap" $\bullet\,\sigma$ and the "fragmental heap" $\circ\,\sigma'$. The fragmental heap is guaranteed to be a subheap of the full one $\sigma' \subseteq \sigma$. Then we make the following definitions:

$$\mathsf{heap\_ctx} \triangleq \boxed{\exists\sigma.\ \mathsf{has\_state}(\sigma) * \boxed{\bullet\,\sigma}} \qquad \ell \mapsto \alpha \triangleq \boxed{\circ\,[\ell \mapsto \mathsf{next}(\alpha)]}$$

The heap_ctx predicate is an invariant that says that the full view of the heap coincides with the actual state that we have as part of has_state, and the points-to predicate $\ell \mapsto \alpha$ states that $[\ell \mapsto \mathsf{next}(\alpha)]$ is in the fragmental view of the heap. Together those predicates imply that the actual state σ maps ℓ to next(α), which is precisely what allows us to deduce the rules WP-ALLOC, WP-READ and WP-WRITE from the rule WP-REIFY.

$$x : \tau, \Gamma \vdash x : \tau \qquad \Gamma \vdash n : \mathsf{Nat} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Nat} \qquad \Gamma \vdash e_2 : \mathsf{Nat} \qquad \oplus \in \{-,+\}}{\Gamma \vdash e_1 \oplus e_2 : N}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{f : \tau_1 \to \tau_2, x : \tau_1, \Gamma \vdash e : \tau_1 \to \tau_2}{\Gamma \vdash \mathsf{rec}\ f(x) = e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e : \mathsf{Nat} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau} \qquad \Gamma \vdash \mathsf{input} : \mathsf{Nat} \qquad \frac{\Gamma \vdash e : \mathsf{Nat}}{\Gamma \vdash \mathsf{output}(e) : \mathsf{Nat}}$$

Fig. 7. Typing rules for $\lambda_{\mathrm{rec,io}}$.

As a simple example, we can use the rules for the store operations to verify the factorial program from Example 3.5. We show the following specification: wp $\mathrm{fact}(n)\ \{\beta_v.\ \beta_v = \mathrm{Ret}(!n)\}$. For this, we will use the intermediate lemma:

Lemma 6.4. *Under the assumptions* heap_ctx, $acc \mapsto \mathrm{Ret}(m)$ *and* $\ell \mapsto \mathrm{Ret}(n)$, *we have*

$$wp\ \mathrm{factBody}(acc, \ell)\ \{\_.\ acc \mapsto \mathrm{Ret}(m \times !n)\}.$$

Proposition 6.5. *Under the assumption* heap_ctx *we have*

$$wp\ \mathrm{fact}(n)\ \{\beta_v.\ \beta_v = \mathrm{Ret}(!n)\}.$$

Proof. We proceed by allocating the locations *acc* and ℓ symbolically using wp-alloc, and then appeal to Lemma 6.4. □

As one can see, the logic that we recovered for the higher-order store effects is very close to a normal separation logic one would normally see for a programming language with a heap [Jung et al. 2018]. Our logic, however, is amenable to extensions with other effects and programming language constructs. Indeed, we explain how to obtain a logic for reasoning about different combined effects in Section 8. In the next section we show how to apply the separation logic to show computational adequacy of the model of $\lambda_{\mathrm{rec,io}}$.

7 COMPUTATIONAL ADEQUACY FOR $\lambda_{\mathrm{rec,io}}$

In Section 5 we constructed a compositional model of $\lambda_{\mathrm{rec,io}}$ in guarded interaction trees and proved that it is sound: if a $\lambda_{\mathrm{rec,io}}$ program e terminates to a natural number n, then $[\![e]\!]$ terminates to $\mathrm{Ret}(n)$. In this section we show the other direction, known as computational adequacy in domain theory [Plotkin 1977], for the well-typed fragment of $\lambda_{\mathrm{rec,io}}$; the typing relation ($\Gamma \vdash e : \tau$) is given in Figure 7. Computational adequacy is formally stated as the following theorem:

Theorem 7.1 (Adequacy). *If* $\vdash e : \mathsf{Nat}$ *and* $([\![e]\!], \sigma) \rightsquigarrow^* (\mathrm{Ret}(n), \sigma')$ *then* $(e, \sigma) \to^* (n, \sigma')$.

Computational adequacy is usually proved using logical relations between the syntax (terms of $\lambda_{\mathrm{rec,io}}$ in our case) and semantics (guarded interaction trees in our case). Here we follow the recent practice [Krebbers et al. 2017b] of using the separation logic (see Section 6) to define our logical relations model.

We define a logical relation $\Gamma \models \alpha \lesssim e : \tau$, relating a guarded interaction tree α and an expression e. Here, e is an open expression for which we have $\Gamma \vdash e : \tau$ while α is "an open interaction tree", i.e., a function of type $(\mathrm{fv}(\Gamma) \to \mathbf{IT}) \to \mathbf{IT}$. As usual, we first define the relation over closed GITrees and expressions, and then generalize it to the open case. The logical relation, given in Figure 8, is, as usual for call-by-value languages, is decomposed into an expression relation $\mathcal{E}[\![\tau]\!]$ and a value

$$\begin{aligned}
\mathcal{E}[\![\tau]\!](\alpha, e) &\triangleq \forall \sigma.\, \mathsf{has\_state}(\sigma) \twoheadrightarrow \mathsf{wp}\, \alpha \left\{\beta_v.\, \exists v,\, \sigma'.\, (e, \sigma) \to^* (v, \sigma') * \mathcal{V}[\![\tau]\!](\beta_v, v) * \mathsf{has\_state}(\sigma')\right\} \\
\mathcal{V}[\![\mathsf{Nat}]\!](\beta_v, v) &\triangleq \exists n \in \mathbb{N}.\, \beta_v = \mathsf{Ret}(n) \wedge v = n \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!](\beta_v, v) &\triangleq \exists f.\, \beta_v = \mathsf{Fun}(f) \wedge \Box\big(\forall \alpha_w,\, w.\, \mathcal{V}[\![\tau_1]\!](\alpha_w, w) \twoheadrightarrow \mathcal{E}[\![\tau_2]\!](\beta_v \bullet \alpha_w, v\, w)\big). \\
\mathcal{V}^*[\![\Gamma]\!](\rho_1, \rho_2) &\triangleq \forall (x : \tau) \in \Gamma.\, \mathcal{V}[\![\tau]\!](\rho_1(x), \rho_2(x)) \\
\Gamma \models \alpha \lesssim e : \tau &\triangleq \forall \rho_1,\, \rho_2.\, \mathcal{V}^*[\![\Gamma]\!](\rho_1, \rho_2) \implies \mathcal{E}[\![\tau]\!](\alpha(\rho_1), e[\rho_2])
\end{aligned}$$

Fig. 8. Logical relation for $\lambda_{\mathsf{rec,io}}$.

relation $\mathcal{V}[\![\tau]\!]$. The expression relation simply states that related expressions should produce related values. The value relation is defined by induction on the type in the standard way: values of base types should be equal while functions take related values to related expressions. As values, once computed, can be used multiple times (cf. the logical relation in Section 9.1) the value relation is required to be persistent; hence the persistently modality $\Box$ in the value relation for functions. In order to define the relation on open terms we define a relation for typing contexts $\mathcal{V}^*[\![\Gamma]\!]$ which relates, point-wise, two substitutions respectively of the types $\mathsf{fv}(\Gamma) \to \mathbf{IT}^v$ and $\mathsf{fv}(\Gamma) \to \mathit{Val}$.

Lemma 7.2 (Fundamental property). *For any* $\Gamma \vdash e : \tau$*, we have* $\Gamma \models (\lambda \rho.\, [\![e]\!]_\rho) \lesssim e : \tau$.

Computational adequacy follows from the fundamental property together with the following Lemma which itself is a consequence of the soundness of the weakest precondition calculus (Theorem 6.2):

Lemma 7.3. *Suppose that* $\models \alpha \lesssim e : \mathsf{Nat}$*. Then for any state* σ*,*

- *if* $(\alpha, \sigma) \rightsquigarrow^* (\mathsf{Ret}(n), \sigma')$*, then* $(e, \sigma) \to^* (n, \sigma')$*;*
- *if* $(\alpha, \sigma) \rightsquigarrow^* (\beta, \sigma')$*, then* $\beta \neq \bot$*.*

8 MODULAR REASONING ABOUT COMBINATIONS OF EFFECTS

Because (guarded) interaction trees define effects abstractly, one of the main advantages is the ability to combine programs with different effects modularly in the same setting. In this section we demonstrate how we achieve this for guarded interaction trees.

Given two signatures E and F, with indexing sets I and J, we say that E is a subsignature of F, written as $E \rightarrowtail F$, if there is a mapping $\epsilon : I \to J$ such that $E.\mathit{Ins}_i(X) \simeq F.\mathit{Ins}_{\epsilon(i)}(X)$ and $E.\mathit{Outs}_i(X) \simeq F.\mathit{Outs}_{\epsilon(i)}(X)$ for any $i \in I$ and for any type X. Here, $\simeq$ stands for isomorphism of types.

In regular interaction trees, a subsignature $E \rightarrowtail F$ induces an embedding $\mathbf{IT}_E \to \mathbf{IT}_F$ of interaction trees. However, such an embedding is not possible for guarded interaction trees due to the mixed-variance definition: a function $\mathbf{IT}_E \to \mathbf{IT}_E$ cannot be converted to a function $\mathbf{IT}_F \to \mathbf{IT}_F$ which takes a guarded interaction tree with a larger set of effects.

To achieve modularity we will instead work with an open-ended collection of effects which is large enough to embed all the effects that we need. It is only at the "top-level", e.g., when applying the adequacy theorem, that we pick a concrete signature of effects. For example, the precise type of the Alloc function from Example 3.2 is the following type, for any F such that $E_{store} \rightarrowtail F$:

$$\mathsf{Alloc} : \mathbf{IT}_F \times (\mathit{Loc} \to \mathbf{IT}_F) \to \mathbf{IT}_F$$

(In practice, the function Alloc is polymorphic in F *at the meta-level*, i.e., in Coq.) With this, we can easily combine two programs with different collections of effects, assuming both of the programs are written in such an open ended manner; we just need to pick F to be large enough to embed the effects of both programs. For example, we can combine the factorial implementation from Example 3.5 with input/output effects, to write a program that takes a natural number from the input, computes its factorial, and prints the result to the output:

$$\mathsf{fact\_io} \triangleq \mathsf{get\_nat}(\mathsf{get\_nat}(\mathsf{Input}, \mathsf{fact}), \mathsf{Output}).$$

The resulting program fact_io has type $\mathbf{IT}_F$ for any F such that $E_{store} \rightarrowtail F$ and $E_{io} \rightarrowtail F$.

Reifiers for modular effects. Writing down programs with modular combinations of effects is not enough by itself: we also want to reason about the reification of effects modularly. Suppose we write a program with effects E as an GITree $\mathbf{IT}_F$ with $E \rightarrowtail F$, and suppose that we have a reifier for E. Recall that we defined a reifier for the effects E to be a tuple $(E, State, r : \prod_{\mathrm{i} \in E} Ins_{\mathrm{i}}(\blacktriangleright \mathbf{IT}_E) \times State \rightarrow option(Outs_{\mathrm{i}}(\blacktriangleright \mathbf{IT}_E) \times State)$. However, if the state itself includes interaction trees, as in Example 4.2, we need also to make the state and the reifier parametric in the effects. Therefore, instead of a fixed type $State$ we consider a family of states $State(X)$, and instead a single reifier function r we consider a family of functions

$$\forall X. \prod_{\mathrm{i} \in E} Ins_{\mathrm{i}}(X) \times State(X) \rightarrow option(Outs_{\mathrm{i}}(X) \times State(X)).$$

In practice, we assume that the global state is the product of states of reifiers for sub-effects, in which each sub-effect acts only on its own part of the state. Concretely, given a sequence $\vec{R} = (E_1, State_1, r_1), \ldots, (E_m, State_m, r_m)$ of reifiers we define the *global reifier* $R_G = (G, State(-), r)$:

$$G.I \triangleq \sum_{1 \le i \le m} E_i.I \qquad G.Ins_{(i,j)}(X) \triangleq E_i.Ins_j(X) \qquad G.Outs_{(i,j)}(X) \triangleq E_i.Outs_j(X)$$

$$State(X) \triangleq \prod_{1 \le i \le m} State_i(X)$$

$$r_{X,(i,j)}(x, (\sigma_1, \ldots, \sigma_i, \ldots, \sigma_m)) \triangleq \begin{cases} \mathsf{Some}(y, (\sigma_1, \ldots, \sigma'_i, \ldots, \sigma_m)) & \text{if } r_i(x, \sigma_i) = \mathsf{Some}(y, \sigma') \\ \mathsf{None} & \text{otherwise} \end{cases}$$

Turning to the separation logic, we specialize the rule WP-REIFY to the signature G and the reifier R_G, and simplify it to

WP-REIFY-LOCAL
$$\frac{\mathsf{has\_state}_i(\sigma_i) \qquad r_i(x, \sigma_i) = \mathsf{Some}(y, \sigma'_i) \qquad k\ y = \mathsf{next}(\beta) \qquad \triangleright\big(\mathsf{has\_state}_i(\sigma'_i) \mathrel{-\!\!*} \mathsf{wp}\ \beta\ \{\Phi\}\big)}{\mathsf{wp}\ \mathsf{Vis}_{(i,j)}(x, k)\ \{\Phi\}}$$

where the predicate $\mathsf{has\_state}_i(\sigma)$ tracks the local component of the global state associated with the i^{th} reifier. The predicates are defined to validate the following rule, which allows us to split the global state into local subcomponents and combine them back together:

$$\mathsf{has\_state}(\vec{\sigma}) \dashv\vdash \mathsf{has\_state}_1(\sigma_1) * \cdots * \mathsf{has\_state}_m(\sigma_m).$$

Then to write down the abstractions for the domain-specific logic in Section 6.2 we change the heap_ctx definition to link together only the state corresponding to the specific effects:

$$\mathsf{heap\_ctx} \triangleq \boxed{\exists \sigma.\, \mathsf{has\_state}_i(\sigma) * \boxed{\bullet\, \sigma}}$$

where the higher-order store reifier is the i^{th} subreifier of $\vec{R}$.

Example 8.1. Recall the program fact_io from the beginning of this section. We use Proposition 6.5 to show the following specification:

$$\mathsf{heap\_ctx} * \mathsf{has\_state}_j(k\vec{n}, \vec{m}) \vdash \mathsf{wp}\ \mathsf{fact\_io}\ \{\_.\ \mathsf{has\_state}_j\ (\vec{n}, (!k)\vec{m})\ \}$$

where $\mathrm{has\_state}_j$ tracks the state of the input/output effects. The specification tells us that if we run fact_io with the starting state $(k\vec{n}, \vec{m})$ for the input/output effects, then we end up with the state $(\vec{n}, (!k)\vec{m})$ for the input/output effects.

Modular reasoning with a generic ground type. As we have mentioned in Section 3.2, we often would like to work with the GITrees $\mathbf{IT}_E(A)$ for some generic ground type A that is "large enough" to contain ground values that we need to represent (e.g. the unit type, natural numbers, the type of locations, etc). That is, we assume that the ground type A is isomorphic to a sum $\mathbf{1} + \mathsf{Nat} + \mathit{Loc} + \ldots$, depending on ground values we need. We tackle this generic ground type in a similar way we deal with different effect signatures modularly.

Specifically, we write $B \rightharpoondown A$ if $A \simeq B + C$ for some type C. We then have the generic return constructor $\mathsf{Ret} : B \longrightarrow \mathbf{IT}_E(A)$ for any $B \rightharpoondown A$. Similarly, we have a generic "destructor" $\mathsf{get\_ret} : \mathbf{IT}_E(A) \times (B \longrightarrow \mathbf{IT}_E(A)) \longrightarrow \mathbf{IT}_E(A)$ which allows us to extract a ground value of type B, as long as we have $B \rightharpoondown A$. such that get_ret is a homomorphism in the first argument, which satisfies:

$$\mathsf{get\_ret}(\mathsf{Ret}(b), g) = \begin{cases} g(b) \text{ if } b \in B, \\ \mathsf{Err}(\mathit{RunTime}) \text{ otherwise.} \end{cases}$$

Then, the get_nat function from Section 3.2 is just the specialization of get_ret to the situation $\mathsf{Nat} \rightharpoondown A$.

The predicate $B \rightharpoondown A$ is formalized in Coq as a typeclass, making it easy to use the generic operations like Ret and get_ret. In the remainder of this paper we will stick to those generalized operations, and will assume that the ground type A contains all the ground values we need.

9 TYPE SAFETY FOR CROSS-LANGUAGE INTEROPERABILITY

One of the advantages of using GITrees for denotational semantics is that it provides a common setting for interpreting and reasoning about different languages with different effects, and then combining the results in a modular manner. In this section we demonstrate this point by verifying type safety of interoperability between two different languages. The interoperability is achieved by allowing embeddings from one language into another at a particular boundary [Matthews and Findler 2007]. We take inspiration for this case study from the approach of Patterson et al. [2022], who consider interoperability of different languages at a level of a common third language, which both the source languages are compiled down to. The communication between the source languages is done through *glue code* at the level of the target language, which converts types from one language to another. The type safety result then states that well-typed programs in a combined language can only go wrong due to conversion errors at the boundaries.

Specifically, we first consider an affine programming language $\lambda_{\multimap,\mathrm{ref}}$ with linear references and strong updates, which we interpret in guarded interaction trees using the higher-order store effects (Example 3.2). We show the type safety of $\lambda_{\multimap,\mathrm{ref}}$ by building a logical relations model.

We then consider cross-language interoperability, by allowing embedding of (higher-order) programs from the non-affine language $\lambda_{\mathrm{rec,io}}$ (Section 5) into $\lambda_{\multimap,\mathrm{ref}}$, thus combining two languages with different type systems and different effects. Following the approach outlined in Section 8 we *reuse the standalone interpretations* of $\lambda_{\mathrm{rec,io}}$ and $\lambda_{\multimap,\mathrm{ref}}$ to interpret the combined language in $\mathbf{IT}_{\mathit{store,io}}$. Finally, we show type safety of this combined language, by building the logical relations model out of the models for the individual languages.

We stress that our approach here allows us to prove type safety of $\lambda_{\mathrm{rec,io}}$ and $\lambda_{\multimap,\mathrm{ref}}$ *separately*, and then prove type safety of the combined language by *reusing* the logical relations for the individual languages, thus highlighting the modular nature of the GITrees framework.

$$\frac{b \in \mathbb{B}}{\Omega \vdash b : \mathsf{Bool}} \qquad \frac{n \in \mathbb{N}}{\Omega \vdash n : \mathsf{Nat}} \qquad \Omega \vdash () : \mathsf{Unit} \qquad \Omega_1, a : \tau, \Omega_2 \vdash a : \tau \qquad \frac{a : \tau_1, \Omega \vdash e : \tau_2}{\Omega \vdash \lambda a.\, e : \tau_1 \multimap \tau_2}$$

$$\frac{\Omega_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Omega_2 \vdash e_2 : \tau_1}{\Omega_1, \Omega_2 \vdash e_1\ e_2 : \tau_2} \qquad \frac{\Omega_1 \vdash e_1 : \tau_1 \qquad \Omega_2 \vdash e_2 : \tau_2}{\Omega_1, \Omega_2 \vdash (e_1, e_2) : \tau_1 \otimes \tau_2}$$

$$\frac{\Omega_1 \vdash e_1 : \tau_1 \otimes \tau_2 \qquad a_1 : \tau_1, a_2 : \tau_2, \Omega_2 \vdash e_2 : \tau}{\Omega_1, \Omega_2 \vdash \mathsf{let}\ (a_1, a_2) = e_1\ \mathsf{in}\ e_2 : \tau} \qquad \frac{\Omega \vdash e : \tau}{\Omega \vdash \mathsf{alloc}(e) : \mathsf{ref}\ \tau} \qquad \frac{\Omega \vdash e : \mathsf{ref}\ \tau}{\Omega \vdash \mathsf{dealloc}(e) : \mathsf{Unit}}$$

$$\frac{\Omega_1 \vdash e_1 : \mathsf{ref}\ \tau_1 \qquad \Omega_2 \vdash e_2 : \tau_2}{\Omega_1, \Omega_2 \vdash \mathsf{replace}(e_1, e_2) : \tau_1 \otimes \mathsf{ref}\ \tau_2}$$

Fig. 9. Type system for $\lambda_{\multimap,\mathsf{ref}}$.

9.1 An Affine Programming Language

First, we consider an affine programming language $\lambda_{\multimap,\mathsf{ref}}$ with references with strong updates, and show how to interpret it in GITrees in a way that enforces linearity. We then consider the combination $\lambda_{\multimap,\mathsf{ref}} + \lambda_{\mathsf{rec,io}}$, which allows us to embed $\lambda_{\mathsf{rec,io}}$ programs, including functions, into $\lambda_{\multimap,\mathsf{ref}}$. The syntax for the affine language $\lambda_{\multimap,\mathsf{ref}}$ is as follows:

$$\begin{aligned} \tau \in \mathit{Type} &::= \mathsf{Bool} \mid \mathsf{Nat} \mid \mathsf{Unit} \mid \tau_1 \otimes \tau_2 \mid \tau_1 \multimap \tau_2 \mid \mathsf{ref}\ \tau \\ e \in \mathit{Expr} &::= n \mid b \mid () \mid a \mid \lambda a.\, e \mid e_1\ e_2 \mid (e_1, e_2) \mid \mathsf{let}\ (a_1, a_2) = e_1\ \mathsf{in}\ e_2 \\ &\quad\ \mid \mathsf{alloc}(e) \mid \mathsf{dealloc}(e_1) \mid \mathsf{replace}(e_1, e_2) \end{aligned}$$

To differentiate between the terms of $\lambda_{\multimap,\mathsf{ref}}$ and the terms of $\lambda_{\mathsf{rec,io}}$, we use the orange color to refer to types and programs of $\lambda_{\multimap,\mathsf{ref}}$. The type system for $\lambda_{\multimap,\mathsf{ref}}$ is given in Figure 9. Let us explain some of the details. The language $\lambda_{\multimap,\mathsf{ref}}$ contains Booleans, natural numbers, and the unit type. It also features linear functions $\tau_1 \multimap \tau_2$. Note that in the typing rule for function application, the context is split between typing of the function and typing of the argument. This ensures that the function and its argument do not share any variables or resources in common.

The language also features linear pairs $\tau_1 \otimes \tau_2$. In the typing rule for pairs the typing environment has to be split between the two components. This ensures that we cannot have, e.g., pairs of the form (x, x).

Finally, the language features references with strong updates, i.e., references that can store values of different types. The constructors alloc and dealloc are used to allocate and free the references, respectively. To ensure linearity, we have a single operation that combines reading from a reference and performing a strong update. The program $\mathsf{replace}(r, v)$ reads the value that is stored in the reference r and updates it to the value v. It then returns a linear pair consisting of the old value and the reference itself, allowing one to reuse the reference later on.

The meaning of $\lambda_{\multimap,\mathsf{ref}}$ is given by the interpretation function $[\![e]\!]_\rho : \mathbf{IT}_F(A)$, where ρ is the environment mapping the free variables of e to GITrees, and where F is a signature which contains the higher-order store effects (Example 3.2). We assume that the ground type A contains, in addition to natural numbers and the unit type, the type Loc of locations. The semantic interpretation of $\lambda_{\multimap,\mathsf{ref}}$ is given in Figure 10. In the interpretation of compound expression we split the environment ρ into the environments ρ_1 and ρ_2, for the free variables of e_1 and e_2 respectively.

$$[\![b]\!]_\rho \triangleq \begin{cases} \mathsf{Ret}(1) & \text{if b} = \mathsf{true} \\ \mathsf{Ret}(0) & \text{otherwise} \end{cases} \qquad [\![()]\!]_\rho \triangleq \mathsf{Ret}(())$$

$$[\![n]\!] \triangleq \mathsf{Ret}(n) \qquad [\![\lambda a.\, e]\!]_\rho \triangleq \mathsf{Fun}(\mathsf{next}(\lambda \alpha.\, [\![e]\!]_{\rho[a \mapsto \alpha]}))$$

$$[\![a]\!]_\rho \triangleq \mathsf{Force}(\rho(a)) \qquad [\![e_1\ e_2]\!]_\rho \triangleq \mathsf{Let}\ x = [\![e_2]\!]_{\rho_2}\ \mathsf{in}\ [\![e_1]\!]_{\rho_1} \bullet \mathsf{Thunk}(x)$$

$$\begin{aligned}[\![\mathsf{let}\ (a_1,a_2) = e_1\ \mathsf{in}\ e_2]\!]_\rho \triangleq\ & \mathsf{Let}\ x = [\![e_1]\!]_{\rho_1}\ \mathsf{in} \\ & \mathsf{Let}\ y = \mathsf{Thunk}(\pi_1\ (x))\ \mathsf{in} \\ & \mathsf{Let}\ z = \mathsf{Thunk}(\pi_2\ (x))\ \mathsf{in} \\ & [\![e_2]\!]_{\rho_2[a_1 \mapsto y, a_2 \mapsto z]}\end{aligned} \qquad [\![(e_1,e_2)]\!]_\rho \triangleq ([\![e_1]\!]_{\rho_2}, [\![e_2]\!]_{\rho_2})$$

$$[\![\mathsf{alloc}(e)]\!]_\rho \triangleq \mathsf{Let}\ x = [\![e]\!]_\rho\ \mathsf{in}\ \mathsf{Alloc}(x, \mathsf{Ret}) \qquad [\![\mathsf{dealloc}(e)]\!]_\rho \triangleq \mathsf{get\_ret}([\![e]\!]_\rho, \mathsf{Dealloc})$$

$$\begin{aligned}[\![\mathsf{replace}(e_1, e_2)]\!]_\rho \triangleq\ & \mathsf{Let}\ y = [\![e_2]\!]_{\rho_2}\ \mathsf{in}\ \mathsf{get\_ret}([\![e_1]\!]_{\rho_1}, \lambda \ell.\, \mathsf{Let}\ x = \mathsf{Read}(\ell)\ \mathsf{in} \\ & \quad \mathsf{Write}(\ell, y)\ ;\ (x, \mathsf{Ret}(n)))\end{aligned}$$

Fig. 10. Interpretation of $\lambda_{\multimap,\mathsf{ref}}$.

In order to ensure that the variables from the context Ω are used at most once, we wrap every variable in a thunk which can be evaluated at most once:

$$\mathsf{Thunk}(\alpha) \triangleq \mathsf{Alloc}\big((0), \lambda \ell.\ \mathsf{Fun}(\mathsf{next}(\lambda\_.\, \mathsf{If}(\mathsf{Read}(\ell), \mathsf{Err}(Lin), \mathsf{Write}(\ell, \mathsf{Ret}(1))\ ;\ \alpha)))\big)$$
$$\mathsf{Force}(\alpha) \triangleq \alpha \bullet \mathsf{Ret}(0)$$

When we called a thunked GITree for the second time, it will return the error $\mathsf{Err}(Lin)$, signifying that we broke the linearity condition. Here we assume that we have a separate error state $Lin \in$ `Error`, because we want to treat linearity condition errors separate from type errors or memory safety errors. As such, in the interpretation of a function application we put the argument in a Thunk, and whenever we use the argument (or any affine variable) we then have to Force it.

We can show that if we have a well-typed program, then it does not have any run-time errors, and that all the thunks are evaluated at most once:

Proposition 9.1. *Suppose that* $\vdash e : \tau$, *and suppose that* $(\sigma, [\![e]\!]) \rightsquigarrow^* (\sigma', \beta)$. *Then* $\beta \neq \mathsf{Err}(err)$.

To prove Proposition 9.1 we use a logical relation, given in Figure 11, defined similarly to the logical relation from Section 7. The interpretation $\mathcal{V}[\![-]\!]$ of the base types cover the appropriate subsets of natural numbers. Reference types are interpreted using the "pointsto" $\ell \mapsto \alpha_v$ proposition, and affine pairs are interpreted component-wise. The main differences to note here are: (1) variables in $\lambda_{\multimap,\mathsf{ref}}$ are interpreted as thunks, and thus we adjust the interpretation $\mathcal{V}^*[\![\Omega]\!]$ to account for that; (2) values in $\lambda_{\multimap,\mathsf{ref}}$ can be used at most once; hence the value interpretation is not persistent, i.e., there is no persistently modality $\square$ in the interpretation of function types.

Lemma 9.2 (Fundamental property). *For any expression e, if* $\Omega \vdash e : \tau$, *then* $\Omega \models \lambda\rho.\, [\![\alpha]\!]_\rho : \tau$.

We prove the fundamental property by induction on the typing derivation. More specifically, for each typing rule we prove an associated *compatibility lemma*, by replacing expressions with interaction trees and the derivability $\vdash$ with validity $\models$. For example, the compatibility lemma for dealloc looks as follows:

Lemma 9.3. *Suppose that* $\Omega \models \alpha : \mathsf{ref}\ \tau$. *Then* $\Omega \models \lambda\rho.\, \mathsf{get\_ret}(\alpha(\rho), \mathsf{Dealloc}) : \mathsf{Unit}$.

$$
\begin{aligned}
\mathcal{V}[\![\mathsf{Unit}]\!](\beta_v) &\triangleq \beta_v = \mathsf{Ret}(()) & \mathcal{V}[\![\mathsf{Nat}]\!](\beta_v) &\triangleq \exists n \in \mathbb{N}.\, \beta_v = \mathsf{Ret}(n)\\
\mathcal{V}[\![\tau_1 \multimap \tau_2]\!](\beta_v) &\triangleq \forall \alpha_w.\, \mathcal{V}[\![\tau_1]\!](\alpha_w) \mathrel{-\!\!*} \mathcal{E}[\![\tau_2]\!](\beta_v \bullet \alpha_w) & \mathcal{V}[\![\mathsf{Bool}]\!](\beta_v) &\triangleq \beta_v = \mathsf{Ret}(0) \lor \beta_v = \mathsf{Ret}(1)\\
\mathcal{V}[\![\mathsf{ref}\ \tau]\!](\beta_v) &\triangleq \exists \ell \in \mathit{Loc}, \alpha_v.\, (\beta_v = \mathsf{Ret}(\ell)) * & \mathcal{V}[\![\tau_1 \otimes \tau_2]\!](\beta_v) &\triangleq \exists \gamma_v, \delta_v.\, \beta_v = (\gamma_v, \delta_v) *\\
&\qquad \ell \mapsto \alpha_v * \mathcal{V}[\![\tau]\!](\alpha_v) & &\qquad \mathcal{V}[\![\tau_1]\!](\gamma_v) * \mathcal{V}[\![\tau_2]\!](\delta_v)\\
\mathcal{E}[\![\tau]\!](\alpha) &\triangleq \mathsf{heap\_ctx} \mathrel{-\!\!*} \mathsf{wp}\ \alpha\ \{\beta_v.\, \mathcal{V}[\![\tau]\!](\beta_v)\} & \mathsf{protected}(\Phi)(\beta_v) &\triangleq \mathsf{wp}\ \mathsf{Force}(\beta_v)\ \{\Phi\}\\
\mathcal{V}^*[\![\Omega]\!](\rho) &\triangleq \forall (a : \tau) \in \Omega.\, \mathsf{protected}(\mathcal{V}[\![\tau]\!])(\rho(a)) & \Omega \models \alpha : \tau &\triangleq \forall \rho.\, \mathcal{V}^*[\![\Omega]\!](\rho) \implies \mathcal{E}[\![\tau]\!](\alpha(\rho))
\end{aligned}
$$

Fig. 11. Logical relation for $\lambda_{\multimap,\mathsf{ref}}$.

Proving all the compatibility lemmas is relatively straightforward using the separation logic rules. Having separate compatibility lemmas will be useful for us in the next section.

By combining the fundamental property with the safety theorem for the weakest precondition calculus (Theorem 6.2) we obtain a proof of Proposition 9.1.

Safety for $\lambda_{\mathsf{rec,io}}$. Similar to the logical relation for safety for $\lambda_{\multimap,\mathsf{ref}}$, we define a logical relation for $\lambda_{\mathsf{rec,io}}$. It is simply an unary version of the logical relation from Section 7. For example, the expression relation is defined as

$$\mathcal{E}[\![\tau']\!](\alpha) \triangleq \forall \sigma'.\, \mathsf{has\_state}_i(\sigma') \mathrel{-\!\!*} \mathsf{wp}\ \alpha\ \{\beta_v.\, \exists \sigma'_1.\, \mathcal{V}[\![\tau']\!](\beta_v) * \mathsf{has\_state}_i(\sigma'_1)\}$$

where $\mathsf{has\_state}_i$ tracks the state for the input/output effects. We omit the other details here and direct an interested reader to the Coq formalization. We only note that just like for $\lambda_{\multimap,\mathsf{ref}}$, we split the proof of the fundamental property into compatibility lemmas, which we will use in the next section.

9.2 Interoperability Between Languages

Following the approach of [Patterson et al. 2022], we allow the interaction between the languages $\lambda_{\mathsf{rec,io}}$ and $\lambda_{\multimap,\mathsf{ref}}$, using the guarded interaction trees as the "common ground", combining the effects of the two languages. At the syntactic language level, this is done by allowing one to embed the expressions of $\lambda_{\mathsf{rec,io}}$ into the $\lambda_{\multimap,\mathsf{ref}}$ programs. The embedding is given by the following typing rule:[4]

$$\textsc{typed-conv}\quad \frac{\vdash e : \tau' \qquad \tau' \sim \tau}{\Omega \vdash (\!| e |\!)_{\tau' \sim \tau} : \tau}$$

The crucial ingredient for the interaction is a type conversion relation $\tau' \sim \tau$ stating that the $\lambda_{\mathsf{rec,io}}$ type τ' is convertible to the $\lambda_{\multimap,\mathsf{ref}}$ type τ.

We have the following type conversions:

$$\mathsf{Nat} \sim \mathsf{Nat} \qquad \mathsf{Nat} \sim \mathsf{Unit} \qquad \mathsf{Nat} \sim \mathsf{Bool} \qquad \frac{\tau'_1 \sim \tau_1 \qquad \tau'_2 \sim \tau_2}{(\mathsf{Nat} \to \tau'_1) \to \tau'_2 \sim \tau_1 \multimap \tau_2}$$

The first three type conversions say that we can freely convert between integers, Booleans, and the unit type (since all of them have similar internal representation). The last conversion is more interesting. It says that we can convert between affine functions and non-affine functions. The affine argument τ_1 is represented as a thunk $(\mathsf{Nat} \to \tau'_1)$, which we will protect at runtime to

[4]For simplicity, we only consider one-way embeddings from $\lambda_{\mathsf{rec,io}}$ to $\lambda_{\multimap,\mathsf{ref}}$, and we only consider embeddings of closed terms. This simplifies the type system, but does not lead to loss of expressivity, since we allow type conversions of functions.

ensure that it is not invoked more than once. The type Nat is used in lieu of the unit type which is absent from $\lambda_{\text{rec,io}}$.

Glue code for conversion functions. In order to (1) convert between different types, and (2) ensure the linearity of $\lambda_{\multimap,\text{ref}}$ programs that might cross the boundary to $\lambda_{\text{rec,io}}$, we need to interpret embedded terms with additional *glue code*. For every type conversion $\tau' \sim \tau$ we generate recursively a pair of conversion functions $C_{\tau',\tau}$ and $C_{\tau,\tau'}$ which convert the representations from τ' to τ and vice versa. The glue code for converting between the base types ensures that the underlying natural number representation stays within the range. For example, for Nat $\sim \mathbb{B}$ we have:

$$C_{\text{Nat},\text{Bool}}(\alpha) \triangleq \text{If}(\alpha, \text{Ret}(1), \text{Ret}(0)) \qquad C_{\text{Bool},\text{Nat}}(\alpha) \triangleq \alpha$$

The glue code for converting functions is a bit more involved:[5]

$$\begin{aligned} C_{(\text{Nat}\to\tau_1')\to\tau_2',\tau_1\multimap\tau_2}(\alpha) &\triangleq \text{Let}\, p = \alpha \,\text{in} \\ &\text{Fun}(\text{next}(\lambda x.\, \text{Let}\, y = C_{\tau_1,\tau_1'}(\text{Force}(x)) \,\text{in}\, C_{\tau_2',\tau_2}(p \bullet \text{Thunk}(y)))) \\ C_{\tau_1\multimap\tau_2,(\text{Nat}\to\tau_1')\to\tau_2'}(\alpha) &\triangleq \text{Let}\, p = \alpha \,\text{in Let}\, p' = \text{Thunk}(p) \,\text{in} \\ &\text{Fun}(\text{next}(\lambda x.\, \text{Let}\, f = \text{Force}(p) \,\text{in Let}\, y = C_{\tau_1',\tau_1}(\text{Force}(x)) \,\text{in}\, C_{\tau_2',\tau_2}(f \bullet \text{Thunk}(y)))) \end{aligned}$$

When we convert a function, we need to recursively convert the argument and the result; in addition the argument to the function needs to be protected with a Thunk. Furthermore, when affine functions are converted to non-affine ones, we need to protect the function itself with a Thunk, to ensure that the function is not invoked multiple times. Calling an affine function multiple times might be unsound, e.g., calling the following function twice will attempt to deallocate an already deallocated reference:

$$(\lambda \ell.\, \lambda\_.\, (\lambda\_.\, 7)\ \text{dealloc}(\ell))\ (\text{alloc}(42)) : \text{Unit}\multimap\text{Nat}.$$

Partial safety for the combined language. We interpret the combined language with embeddings using the glue code. The embedding from $\lambda_{\text{rec,io}}$ to $\lambda_{\multimap,\text{ref}}$ is interpreted as follows:

$$[\![(\!| e |\!)_{\tau'\sim\tau}]\!]_\rho = C_{\tau',\tau}(IO[\![e]\!]),$$

where $IO[\![-]\!]$ is the interpretation function for $\lambda_{\text{rec,io}}$ expressions from Section 5. The interpretation for all the operations, except for the embedding, remains unchanged.

Using the modular approach we described in Section 8, we interpret the extended language $\lambda_{\text{rec,io}} + \lambda_{\multimap,\text{ref}}$ into the guarded interaction trees $\mathbf{IT}_{store,io}$ with the signature that combines the input/output effects and the store effects. This ensures that the interpretation of the languages end up in the same domain, where they can interact. By combining the reifiers for the effects of the individual languages we also get the reduction relation $((\sigma_1, \sigma_1'), \alpha) \rightsquigarrow ((\sigma_2, \sigma_2'), \beta)$, where σ_1 and σ_2 are stores, and σ_1' and σ_2' are input/output tapes.

Of course, in the combined language with conversions, our programs can actually violate the linearity condition, since it is no longer enforced by the type system. However, we can prove that linearity violations at the boundary are the only errors that we will possibly get. Thus we will to prove the following safety theorem:

Theorem 9.4. *Suppose that* $\vdash e : \tau$ *with the embedding rule, and suppose that* $((\sigma_1, \sigma_1'), [\![e]\!]) \rightsquigarrow^* ((\sigma_2, \sigma_2'), \beta)$. *Then either* β *is not an error, or* $\beta = \text{Err}(Lin)$.

[5]The glue code for functions is a bit more complicated than in [Patterson et al. 2022]. Specifically, they do not protect the converted affine functions. This is fine in their setting, because their affine functions are pure, and invoking them multiple times does not lead to run-time errors. However, in the presence of references with strong updates this assumption is no longer true, and not protecting the converted functions will lead to unsound behavior!

9.3 Logical Relations for the Combined Safety

We will prove Theorem 9.4 by constructing a logical relation similarly to what we did for the individual languages in Section 9.1. Our goal is to do so modularly, by reusing as much material from Section 9.1 as possible. In particular, we will reuse all the compatibility lemmas we used to prove Lemma 9.2, and only prove one (!) new compatibility lemma for TYPED-CONV. Letting $\mathcal{E}[\![\tau']\!]$ and $\mathcal{E}[\![\tau]\!]$ be the expression relations for the logical relation for $\lambda_{\mathsf{rec,io}}$ and $\lambda_{\multimap,\mathsf{ref}}$ resp., this compatibility lemma is:

Lemma 9.5. *Suppose that $\mathcal{E}[\![\tau']\!](\alpha)$ and $\tau' \sim \tau$. Then $\mathcal{E}[\![\tau]\!](C_{\tau',\tau}(\alpha))$. Moreover, for the other direction, suppose that $\mathcal{E}[\![\tau]\!](\alpha)$ and $\tau' \sim \tau$. Then $\mathcal{E}[\![\tau']\!](C_{\tau,\tau'}(\alpha))$.*

When we presented the separation logic and logical relations earlier, we presented a slightly simplified version which was sufficient for our purposes. However, in order to prove Lemma 9.5 we need to make use of features that we have not yet presented. We describe those features now.

Separation logic for weak safety. The first feature is that our notion of weakest precondition is actually parameterized by a *stuckness predicate* $s : \mathtt{Error} \rightarrow \mathsf{iProp}$, and satisfies the following rule (in addition to the rules presented in Section 6):

$$s(e) \vdash \mathsf{wp}_s\, \mathsf{Err}(e)\, \{\Phi\}$$

This means that if the stuckness predicate s holds for some error e, then the weakest precondition predicate holds for that error, irrespectively of the postcondition. The earlier presented weakest precondition, which did not allow for errors, is obtained by using $s(e) = \mathsf{False}$. The general weakest precondition with the stuckness predicate satisfies a version of the adequacy/safety property (Theorem 6.2), in which the *(safety)* condition is replaced with the following condition:

- *(weak safety)* either there are β_1 and σ_1 such that $(\beta, \sigma') \rightsquigarrow (\beta_1, \sigma_1)$, or $\beta = \mathsf{Err}(e)$ with e satisfying the predicate s.

All the logical relations presented earlier are actually parameterized by a predicate s and uses $\mathsf{wp}_s\, \alpha\, \{\Phi\}$ in the expression interpretation — to recover the earlier stated theorems for full safety we simply instantiate the logical relations with $s = \lambda e.\, \mathsf{False}$.

Freely adjoined Kripke structure. In addition to the stuckness bit, we actually formulate our logical relations a bit more generally than what we have shown so far. This is because in the logical relations for individual languages require particular resources that we need to combine when constructing a logical relation for the combined language.

Indeed, to ensure that our logical relations are sufficiently modular, we parameterize the expression relation by an arbitrary predicate $P : A \rightarrow \mathsf{iProp}$ of an arbitrary type A. We refer to this predicate P as freely adjoined Kripke structure (because, in the underlying model of Iris, it allows us to make arbitrary transitions between worlds, constrained by the predicate P). The general definition of the expression relation for all our logical relation is thus:

$$\mathcal{E}^P_s(\Phi)(\alpha) \triangleq \forall x : A.\, P(x) \twoheadrightarrow \mathsf{wp}_s\, \alpha\, \{\beta_v.\, \exists y : A.\, \Phi(\beta_v) * P(y)\}.$$

The idea is that the P parameter describes additional resources (for other effects) and ensures that ITrees in the expression relation preserve any such additional resources. The idea is akin to the "baking-in" of the frame rule in models of separation logic for higher-order languages [Birkedal et al. 2008; Birkedal and Yang 2008].

The expression relations for individual languages $\lambda_{\text{rec,io}}$ and $\lambda_{\multimap,\text{ref}}$ are then both parameterized by predicates $P : A \to \text{iProp}$ and $s : \texttt{Error} \to \text{iProp}$ and defined as

$$\mathcal{E}[\![\tau']\!](\alpha) \triangleq \mathcal{E}_s^{\lambda(\sigma',x).\,\text{has\_state}_i(\sigma') * P(x)}(\mathcal{V}[\![\tau']\!])(\alpha)$$
$$\mathcal{E}[\![\tau]\!](\alpha) \triangleq \mathcal{E}_s^{\lambda x.\,\text{heap\_ctx} * P(x)}(\mathcal{V}[\![\tau]\!])(\alpha).$$

For $\lambda_{\text{rec,io}}$ and for $\lambda_{\multimap,\text{ref}}$ we prove the compatibility properties for arbitrary P and s (in the proofs of the compatibility lemmas, the resources described by P are just passed through). We obtain full safety for the individual languages by instantiating P with $\lambda x.\,\text{True}$ and s with $\lambda e.\,\text{False}$.

When we prove partial safety for the combined language, we reuse the same logical relations and the same compatibility lemmas for the individual languages, by instantiating P in $\mathcal{E}[\![\tau']\!]$ with $P(x) = \text{heap\_ctx}$, and by instantiating P in $\mathcal{E}[\![\tau]\!]$ with $P(x) = \text{has\_state}_i(x)$, and s with $\lambda e.\,(e = \mathit{Lin})$ in both cases. Then, to prove the fundamental property of the combined language we can reuse the compatibility lemmas for individual languages, and it only remains to prove the compatibility Lemma 9.5 for the type conversion.

The most interesting case of Lemma 9.5 is the conversion between functions, which involves showing soundness of the glue code. The interpretation of non-affine functions is persistent, as it begins with $\Box$, since it is expected that you can use non-affine functions multiple times. The interpretation of affine functions, however, is not persistent — functions can be invoked only once. Because of that, we cannot directly use the interpretation $\mathcal{V}[\![\tau_1 \multimap \tau_2]\!]$ when proving $\mathcal{V}[\![(\text{Nat} \to \tau_1') \to \tau_2']\!]$. Instead, we put the interpretation of the affine function in a persistent invariant, which states:

$$(\ell \mapsto \text{Ret}(0) * \mathcal{V}[\![\tau_1 \multimap \tau_2]\!](\alpha_v)) \vee (\ell \mapsto \text{Ret}(1)).$$

This describes when the affine function α_v is protected with the Thunk via a reference ℓ: either the function has not been invoked yet ($\ell \mapsto \text{Ret}(0)$) and it satisfies the value interpretation, or the function has already been invoked ($\ell \mapsto \text{Ret}(1)$) and forcing its thunk will result in $\text{Err}(\mathit{Lin})$, in which case we do not invoke the function. (As a side remark, we note that we here crucially rely on Iris's powerful notion of invariants — this is another example of why it is advantageous to use Iris as the basis for our separation logic.)

Having established the compatibility lemma for type conversions, and the compatibility lemmas for the operations for each individual language, we prove the fundamental property for the logical relation for the combined language. In particular, for a closed term $\vdash e : \tau$ we obtain $\mathcal{E}_{\lambda e.\,e=\mathit{Lin}}^{\lambda x.\,\text{heap\_ctx} * \text{has\_state}_i(x)}(\mathcal{V}[\![\tau]\!])([\![e]\!])$. From the adequacy property of the weakest precondition we can then conclude Theorem 9.4.

In summary, this approach to logical relations, with freely adjoined Kripke structure as a parameter, allows us to scale proofs to interoperability between multiple different languages in a modular way. For each individual language we can prove safety separately (without knowing in advance with which other languages we are going to interface). Then we can combine logical relations for individual languages together, by instantiating the freely adjoined Kripke structure with the shared resources or with resources needed to verify the glue code, and reusing the compatibility lemmas.

10 DISCUSSION AND RELATED WORK

We have already discussed a lot of related work throughout the paper; in this section we include some further discussion of related work.

Differences with interaction trees. Whilst our work takes direct inspiration from the interaction trees approach, there are some crucial differences. One of the main difference comes from the treatment of effect reification. The original type of interaction trees is a monad, and the effects in an interaction tree can be reified "in one go", for example, with a state monad transformer over ITrees. In our case, we cannot reify all the effects in a guarded interaction tree, due to higher-order functions and higher-order effects. For example, a guarded interaction tree can be a function that contains latent effects, but these effects can only be reified at the point when the function is invoked. Because regular interaction trees contain only first-order structures, it is possible to traverse them completely, reifying all the effects.

Another difference worth mentioning, is that regular interaction trees can be extracted and executed from Coq. Our formalization cannot be directly extracted, as it is built upon a layer of guarded type theory. One potential approach would be to erase the guarded parts from the formulation of GITrees and obtain that way a representation of GITrees in a functional language like OCaml or Haskell, which already supports mixed-variance datatypes. Then, the extraction can be set up in such a way as to use this representation. We have not researched this direction and leave it to future work.

Finally, an important difference between our work and that on ITrees, is that the ITrees development relies heavily on the weak bisimilarity theory of interaction trees, while we opt for developing separation logic and refinements instead. There are several things that complicate the study of bisimilarity for guarded interaction trees. Firstly, the higher-order nature of GITrees suggests that we need to study a more involved notion of behavioral equivalence, like applicative bisimilarity. Secondly, we believe that developing a theory of *weak* bisimilarity in the context of Iris and guarded type theory is still an open question, complicated by issues with step-indices. For these reasons we believe that developing weak applicative bisimularity for guarded interaction trees will require new techniques and we leave it for future work.

Differences with the standard Iris approach to verification. The standard Iris-based approach to separation logic [Jung et al. 2016, 2015; Krebbers et al. 2017a] is based on operational semantics, and has been proven to scale well to complicated programming language features. The main difference with our work, is that we are the first to build an Iris-based separation logic over denotational semantics, in a way that is tightly integrated with the existing Iris ecosystem. In particular, we rely on the Iris ecosystem for various data structures, resource algebras, base logic (but not the program logic), and the Iris Proof Mode. As such, in terms of reasoning about specific concrete programs, the GITrees-based approach is not that different from what a normal Iris user expects, with the added advantage of using equational reasoning for many computation steps that usually requires some form of symbolic execution.

The main advantage of our approach comes into play when we want to consider new models of programming languages, or reasoning about programs with combinations of effects (as in Section 8), or reasoning about interoperability (as in Section 9).

Domain theory and guarded type theory. Guarded type theory [Birkedal and Møgelberg 2013; Bizjak et al. 2016] has been studied as a setting for domain theory before [Birkedal et al. 2012; Møgelberg and Paviotti 2019; Møgelberg and Vezzosi 2021; Paviotti et al. 2015], but previous works mostly focused on specific typed models and was not formalized. In contrast, here we work with guarded interaction trees as a "universal domain", similar to domain theoretic models of untyped λ-calculus.

The previous work used (dependent) guarded type theory not just for modeling, but also for reasoning about programs. This required a more complicated type theory and precluded the work from being formalized in a traditional proof assistant like Coq. By contrast, our reasoning is done in the logic *over* a guarded type theory. This is arguably simpler, and allowed us to make use of Iris and formalize all of our results in Coq.

Logical relations and language interoperability. Our case study on language interoperability in Section 9 is inspired by the seminal work of Patterson et al. [2022]. We believe that the approach we take in our work is more modular. Firstly, our approach here is less syntactic, as we use the domain theory of guarded interaction trees as the common setting for interpreting different languages with different effects, and we do not need to come up with a target language for each pair of source languages for which we wish to set up interoperability. Secondly, the models that we construct for individual languages are "local", and that is exactly what allows for true reuse of proofs and for constructing a common model for the combined language from the individual models. This opens up the possibility of a model for a single language to be reused for different cross-language interactions. In contrast, the type safety of individual source languages in [Patterson et al. 2022] requires to have a model for the combined language in advance. And finally, the treatment of effects and step-indices is more abstract and high-level in our work, since we construct our models using separation logic.

(Guarded) Interaction Trees and Algebraic Effects and Handlers. The treatment of effects in interaction trees is reminiscent of effects in programming languages based on algebraic effects and handlers [Bauer and Pretnar 2015; Plotkin and Pretnar 2013]. Algebraic effects and handlers have been extensively studied in various contexts, including separation logic [de Vilhena and Pottier 2021], and both higher-order effects [Bach Poulsen and van der Rest 2023; van den Berg et al. 2021; Wu et al. 2014] and reasoning about combinations of effects [Johann et al. 2010; Yang and Wu 2021] have been investigated. Despite the aesthetic and moral similarities to (guarded) interaction trees, there are some substantial differences between the two approaches. Under algebraic effects and handlers, both the representation and reification of effects is done *inside* the programming language. As such, a particular theory and implementation of algebraic effects is always tied to a specific programming language. Whereas in the interaction trees-based approach, the effects are handled in the ambient type theory, outside the type of the (guarded) interaction trees itself. See also the discussion in [Xia et al. 2019, Section 8.2].

To our knowledge, the two approaches have not been formally compared. It would be interesting to consider a denotational model of a programming language with algebraic effects inside guarded interaction trees, and to see what kind of properties can be proved using such a model.

Additionally, such a comparison might help us understand the exact class of effects that can be represented with the GITrees-based approach. As it currently stands, our approach to representing effects is "open-ended", in the sense that we can consider different classes of effects by varying the reification procedure. Of course, different classes of effects allow for different reasoning principles. For example, as we mentioned in the end of Section 4, we consider context-independent effects, which preclude us from modeling call/cc, but allows us to use the bind rule for the weakest precondition calculus. We leave a formal comparison with algebraic effects and further investigations in that area to future work.

ACKNOWLEDGMENTS

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. The authors are grateful to the anonymous reviewers for their comments and suggestions.

DATA AVAILABILITY STATEMENT

The Coq formalization corresponding to this article is available as a Git repository at https://github.com/logsem/gitrees/tree/popl24 (tag `popl24`), or under a permanent DOI at https://doi.org/10.5281/zenodo.10124427.

REFERENCES

Casper Bach Poulsen and Cas van der Rest. 2023. Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 62:1801–62:1831. https://doi.org/10.1145/3571255

Andrej Bauer and Matija Pretnar. 2015. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (Jan. 2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

Lars Birkedal and Rasmus Ejlers Møgelberg. 2013. Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. IEEE Computer Society, 213–222. https://doi.org/10.1109/LICS.2013.27

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Log. Methods Comput. Sci.* 8, 4 (2012). https://doi.org/10.2168/LMCS-8(4:1)2012

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, and Hongseok Yang. 2008. A Simple Model of Separation Logic for Higher-Order Store. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations (Lecture Notes in Computer Science, Vol. 5126)*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.). Springer, 348–360. https://doi.org/10.1007/978-3-540-70583-3_29

Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. 2010. The Category-Theoretic Solution of Recursive Metric-Space Equations. *Theoretical Computer Science* 411, 47 (Oct. 2010), 4102–4122. https://doi.org/10.1016/j.tcs.2010.07.010

Lars Birkedal and Hongseok Yang. 2008. Relational Parametricity and Separation Logic. *Log. Methods Comput. Sci.* 4, 2 (2008). https://doi.org/10.2168/LMCS-4(2:6)2008

Ales Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg, and Lars Birkedal. 2016. Guarded Dependent Type Theory with Coinductive Types. *CoRR* abs/1601.01586 (2016). arXiv:1601.01586 http://arxiv.org/abs/1601.01586

Paulo Emílio de Vilhena and François Pottier. 2021. A Separation Logic for Effect Handlers. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 33:1–33:28. https://doi.org/10.1145/3434314

Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science* 103, 2 (1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

Peter Freyd. 1991. Algebraically Complete Categories. In *Category Theory (Lecture Notes in Mathematics)*, Aurelio Carboni, Maria Cristina Pedicchio, and Guiseppe Rosolini (Eds.). Springer, Berlin, Heidelberg, 95–104. https://doi.org/10.1007/BFb0084215

Naohiko Hoshino. 2012. Step Indexed Realizability Semantics for a Call-by-Value Language Based on Basic Combinatorial Objects. In *2012 27th Annual IEEE Symposium on Logic in Computer Science*. 385–394. https://doi.org/10.1109/LICS.2012.74

Patricia Johann, Alex Simpson, and Janis Voigtländer. 2010. A Generic Operational Metatheory for Algebraic Effects. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*. IEEE Computer Society, 209–218. https://doi.org/10.1109/LICS.2010.29

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 256–269. https://doi.org/10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 234–248. https://doi.org/10.1145/3293880.3294106

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. https://doi.org/10.1145/3009837.

3009855

Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. 2022. C4: Verified Transactional Objects. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 80:1–80:31. https://doi.org/10.1145/3527324

Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. *ACM SIGPLAN Notices* 42, 1 (Jan. 2007), 3–10. https://doi.org/10.1145/1190215.1190220

Rasmus E. Møgelberg and Marco Paviotti. 2019. Denotational Semantics of Recursive Types in Synthetic Guarded Domain Theory. *Mathematical Structures in Computer Science* 29, 3 (March 2019), 465–510.

Rasmus Ejlers Møgelberg and Andrea Vezzosi. 2021. Two Guarded Recursive Powerdomains for Applicative Simulation. *Electronic Proceedings in Theoretical Computer Science* 351 (Dec. 2021), 200–217. arXiv:2112.14056 [cs]

Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic Soundness for Language Interoperability. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 609–624. https://doi.org/10.1145/3519939.3523703

Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal. 2015. A Model of PCF in Guarded Type Theory. *Electronic Notes in Theoretical Computer Science* 319 (Dec. 2015), 333–349.

Andrew M. Pitts. 1996. Relational Properties of Domains. *Information and Computation* 127, 2 (June 1996), 66–90. https://doi.org/10.1006/inco.1996.0052

G. D. Plotkin. 1977. LCF Considered as a Programming Language. *Theoretical Computer Science* 5, 3 (Dec. 1977), 223–255. https://doi.org/10.1016/0304-3975(77)90044-5

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* Volume 9, Issue 4 (Dec. 2013). https://doi.org/10.2168/LMCS-9(4:23)2013

Dana Scott. 1976. Data Types as Lattices. *SIAM J. Comput.* 5, 3 (Sept. 1976), 522–587. https://doi.org/10.1137/0205037

Lucas Silver, Paul He, Ethan Cecchetti, Andrew K Hirsch, and Steve Zdancewic. 2023. Semantics for Noninterference with Interaction Trees. (2023).

M. B. Smyth and G. D. Plotkin. 1982. The Category-Theoretic Solution of Recursive Domain Equations. *SIAM J. Comput.* 11, 4 (Nov. 1982), 761–783. https://doi.org/10.1137/0211062

Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. 2021. Latent Effects for Reusable Language Components. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Hakjoo Oh (Ed.). Springer International Publishing, Cham, 182–201. https://doi.org/10.1007/978-3-030-89051-3_11

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2633357.2633358

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 51:1–51:32. https://doi.org/10.1145/3371119

Zhixuan Yang and Nicolas Wu. 2021. Reasoning about Effect Interaction by Fusion. *Proceedings of the ACM on Programming Languages* 5, ICFP (Aug. 2021), 73:1–73:29. https://doi.org/10.1145/3473578

Kangfeng Ye, Simon Foster, and Jim Woodcock. 2022. Formally Verified Animation for RoboChart Using Interaction Trees. In *Formal Methods and Software Engineering (Lecture Notes in Computer Science)*, Adrian Riesco and Min Zhang (Eds.). Springer International Publishing, Cham, 404–420. https://doi.org/10.1007/978-3-031-17244-1_24

Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, Compositional, and Executable Formal Semantics for LLVM IR. *Proceedings of the ACM on Programming Languages* 5, ICFP (Aug. 2021), 67:1–67:30. https://doi.org/10.1145/3473572

Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, William Mansky, Benjamin Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 32:1–32:19. https://doi.org/10.4230/LIPIcs.ITP.2021.32

Received 2023-07-11; accepted 2023-11-07