Context-Dependent Effects and Concurrency in Guarded Interaction Trees

SERGEI STEPANENKO, Aarhus University, Denmark
EMMA NARDINO, ENS Lyon, France
VIRGIL MARIONNEAU, ENS Rennes, France
DAN FRUMIN, University of Groningen, The Netherlands
AMIN TIMANY, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

Guarded Interaction Trees are a structure and a fully formalized framework for representing higher-order computations with higher-order effects in Rocq. We present an extension of Guarded Interaction Trees to support formal reasoning about context-dependent effects. That is, effects whose behaviors depend on the evaluation context, *e.g.*, `call/cc`, `shift` and `reset`. Using and reasoning about such effects is challenging since certain compositionality principles no longer hold in the presence of such effects. For example, the so-called "bind rule" in modern program logics (which allows one to reason modularly about a term inside an evaluation context) is no longer valid. The goal of our extension is to support representation and reasoning about context-dependent effects in the most painless way possible. To that end, our extension is conservative: the reasoning principles (and the Rocq implementation) for context-independent effects remain the same. We show that our implementation of context-dependent effects is viable and powerful. We use it to give direct-style denotational semantics for higher-order programming languages with `call/cc` and with delimited continuations. We extend the program logic for Guarded Interaction Trees to account for context-dependent effects, and we use the program logic to prove that the denotational semantics is adequate with respect to the operational semantics. This is achieved by constructing logical relations between syntax and semantics inside the program logic. Additionally, we retain the ability to combine multiple effects in a modular way, which we demonstrate by showing type soundness for safe interoperability of a programming language with delimited continuations and a programming language with higher-order store. Furthermore, as another contribution, in addition to context-dependent effects, we show how to extend Guarded Interaction Trees with preemptive concurrency. To support implementation and verification of concurrent data structures and algorithms in the presence of preemptive concurrency one requires atomic state modification operations, *e.g.*, compare-and-exchange. We demonstrate how the ability to pass lambda functions as effect inputs allows us to define a single generic atomic state modification effect from which various atomic operations can be derived.

CCS Concepts: • **Theory of computation** → *Logic and verification*; **Program semantics**.

Additional Key Words and Phrases: Rocq, Iris, denotational semantics, logical relations, control flow operators, continuations, delimited continuations, concurrency

Authors' addresses: Sergei Stepanenko, sergei.stepanenko@cs.au.dk, Aarhus University, Aarhus, Denmark; Emma Nardino, emma.nardino@ens-lyon.fr, ENS Lyon, Lyon, France; Virgil Marionneau, virgil.marionneau@ens-rennes.fr, ENS Rennes, Bruz, France; Dan Frumin, d.frumin@rug.nl, University of Groningen, Groningen, The Netherlands; Amin Timany, timany@cs.au.dk, Aarhus University, Aarhus, Denmark; Lars Birkedal, birkedal@cs.au.dk, Aarhus University, Aarhus, Denmark.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 0164-0925/2025/10-ART
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Sergei Stepanenko, Emma Nardino, Virgil Marionneau, Dan Frumin, Amin Timany, and Lars Birkedal. 2025. Context-Dependent Effects and Concurrency in Guarded Interaction Trees. *ACM Trans. Program. Lang. Syst.* 1, 1 (October 2025), 39 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Despite a lot of recent progress, representing and reasoning about programming languages in proof assistants, such as Rocq, is still considered a major challenge. The design space is wide and many approaches have been considered. On one end, denotational models grounded in Scott domain theory provide compositional, syntax-independent semantics and combine cleanly with monads to structure computational effects [28]; mechanizing such models in a proof assistant has, however, met with only limited success, as faithfully capturing features such as recursion, higher-order store, and control is technically demanding [5]. Partly for this reason, much subsequent work instead builds on operational semantics, where syntactic techniques such as (step-indexed) logical relations scale well to the metatheory of realistic languages [38]. More recently, a complementary line of work has pushed the denotational approach by representing possibly-diverging effectful computations coinductively, while remaining executable and amenable to mechanization, beginning with Capretta's delay monad [10]. Recently, research on a novel point in the design space was initiated with the introduction of Interaction Trees [41], or ITrees for short. ITrees were introduced to simplify representation and reasoning about possibly non-terminating programs with side effects in Rocq. In a sense, ITrees provide a target for denotational semantics of programming languages, which allows one to abstract from syntactic details often found in models based on operational semantics. ITrees specifically allow one to easily represent and reason about various effects and their combinations in a modular way. A wide range of subsequent applications of ITrees (see, *e.g.*, Koh et al. [23], Silver et al. [33], Zakowski et al. [42], Zhang et al. [43], among others) show that they indeed work very well for representing and reasoning about first-order programs with first-order effects. As part of the trade-offs, ITrees could not support higher-order representations and higher-order effects. To address this challenge, Guarded Interaction Trees (or GITrees for short) were introduced [18].

While GITrees support *higher-order effects*, *e.g.*, higher-order store which is known to be particularly challenging, they are limited to effects that do not alter the control flow of the program, *i.e.*, effects that do no modify *the continuation*. As a consequence, one cannot use GITrees to give direct-style denotational semantics of programming languages with context-dependent effects such as `call/cc` [17], exceptions, or delimited continuations [12, 13]. It is of course possible to give semantics to, *e.g.*, `call/cc` using a CPS translation, but this would require a global transformation which complicates representation and reasoning, especially in combination with other effects present. In this paper we extend GITrees to support direct-style representation and reasoning about higher-order programs with higher-order context-dependent effects in Rocq, and evaluate its modularity.

We want to stress that our extension to context-dependent effects is not only theoretically interesting, but also important for scalability, since many real mainstream programming languages include context-dependent effects. Indeed, exceptions are now a standard feature in many languages, and while other context-dependent effects such as delimited continuations are not as widespread in mainstream programming languages, they are present in the core calculi used for some such languages: for example, the Glasgow Haskell Compiler core language was recently extended with delimited continuations to support the introduction of effect systems, which can be efficiently developed on top of delimited continuations [21]. Moreover, effect handlers [31], which rely on control flow operators, have recently been introduced in OCaml 5.0, and as a design feature in

Helium [8], Koka [25, 26], and other languages. Note that these languages do not only include forms of delimited continuations, but also other effects, which underscores the importance of considering delimited control in combination with other effects.

Overview of the technical development and key challenges. Similarly to how ITrees are defined as a coinductive type in Rocq, GITrees are defined as a guarded recursive type. (This is to support function spaces in GITrees, because in the presence of function spaces there are negative occurrences of the recursive type and hence one cannot simply define the type of GITrees using coinduction [18]). Rocq does not directly support guarded recursive types, so GITrees are defined using a fragment of guarded type theory implemented in Rocq as part of the Iris framework [19]. To work efficiently with GITrees we make use of other Iris features like separation logic and Iris Proof Mode [24], which we use to define custom program logics for different (combinations of) effects. This enables us to reason about GITrees smoothly in the Iris logic in Rocq, in much the same way as one works directly in Rocq. We recall the precise definition of GITrees in Section 2.

Broadly speaking, GITrees model effects in the following way. The type of GITrees is parameterized over a set of effectful operations. Each operation is given meaning by a *reifier* function, using a form of state monad. From this, we define the *reduction* relation of GITrees, which gives semantics to computations represented by GITrees.

To support context-dependent effects, we extend (see Section 3) the notion of a reifier so that reification of effects can also depend on the context; technically, the reifier operation becomes parameterized by a suitable GITree continuation. This extension allows us to give semantics to context-dependent effects, but it comes at a price. In particular, following the change in semantics, we need to reformulate the program logic for GITrees: in the presence of context-dependent effects (like `call/cc`), the so-called "bind"-rule becomes unsound. Of course, we do not want this reformulation to complicate reasoning about computations that do *not* include context-dependent effects. To that end, we parameterize the GITrees (and the program logic) by a flag, which allows us to recover the original proof rules and make sure that all of the original GITrees framework still works with our extension.

To motivate the extension to context-dependent effects, we give direct-style denotational semantics to a higher-order programming language, λ_{callcc} with `call/cc` (see Section 3 for details). Furthermore, we use the derived program logic to construct a logical relation between the denotational and the operational semantics to prove computational adequacy of our model. To provide guidance on modeling context-dependent effects, we present exceptions as a pedagogical example (Section 4), showing step-by-step how to design effects, maintain appropriate state (a handler stack), and derive program logic rules.

Our main interest, however, lies in the treatment of delimited continuations. In Section 5 we show how to represent delimited continuations as effects in GITrees, and we use them to define a novel denotational semantics for a programming language with `shift` and `reset` operators. We prove that our denotational semantics is sound with respect to the operational semantics (given by an extension of the CEK abstract machine [17]). We additionally use the program logic to define a logical relation, and prove computational adequacy and *semantic* type soundness. We recall that semantic type soundness is interesting because it allows one to combine syntactically well-formed programs with syntactically ill-typed, but semantically well-behaved programs [38].

As we mentioned, it is important to consider delimited continuations not only on their own, but in combinations with other effects. And indeed, one of the key points of ITrees, and therefore also of GITrees, is that they support reasoning about effecting and language interoperability by establishing a common unifying semantic framework. In this paper, we consider (in Section 6) an example of such interaction: we show a type-safe embedding of λ_{delim}, a language with delimited continuations

into a language λ_{embed}, a language with higher-order store. We allow λ_{delim} expressions to be embedded into λ_{embed} by surrounding them by simple glue code, and use a type system to ensure type safety of the combined language. To define the semantics of the combined language we rely on the modularity of GITrees, and combine reifers for delimited continuations with reifiers for higher-order store. We prove type safety of the combined language by constructing a logical relation and use the program logic both to define the logical relation and to verify the glue code between the two languages. The type system for the combined language naturally requires that the embedded code is well-typed according to the type system for λ_{delim} and thus we can rely on the type soundness of λ_{delim} (proved in the earlier Section 5) when proving type safety for the combined language. At the end of Section 6, we give an example of how to verify a more involved interaction of effects, albeit without the type system.

Additionally, we demonstrate the flexibility of GITrees by presenting another extension of GITrees, preemptive concurrency (Section 7), which is orthogonal to the extension with context-dependent effects. The key insight is that GITrees' support for higher-order effects, specifically, effects that take lambda functions as inputs, enables a generic atomic state modification effect. This is essential for correctly implementing shared mutable state in a concurrent setting, as we demonstrate by presenting denotational semantics for a language featuring fork and atomic exchange operations.

Summary of Contributions. In summary, we present:

- A conservative (with respect to the old results) extension to GITrees for representing and reasoning about context-dependent effects (Section 3).
- A sound and adequate model of a calculus with `call/cc` and `throw`, implemented in a direct style (Section 3.3).
- A tutorial-style presentation of modeling exceptions, providing step-by-step guidance on designing context-dependent effects in GITrees (Section 4).
- A sound and adequate model of a calculus with delimited continuations, with operations `shift` and `reset`, implemented in a direct style (Section 5).
- A type system for interoperability between a programming language with delimited continuations and a programming language with higher-order store, with a semantic type safety proof (Section 6).
- An extension of GITrees with preemptive concurrency, demonstrating how lambda functions in effects enable generic atomic operations, with a sound model of an affine language with fork and and runtime linearity checks using atomic exchange (Section 7).

All results in the paper have been formalized in Rocq as a modification to the GITrees library and the previously proved results have been ported to our extension. In particular, our extension is conservative: we recover the original case studies, computational adequacy of the IO language and type safety of the affine language [18], with minimal modifications (Section 3.2). The full mechanized development is publicly available at https://github.com/logsem/gitrees. We conclude and discuss related work in Section 8. Before we go on with the main part of the paper, we recall some background material on GITrees.

Contributions Compared to the ESOP'25 [35] Version of this Paper. Most of this publication is an almost verbatim repetition of the earlier publication [35] with minor adjustment and updates, *e.g.*, fixing typos, making the text, including this Introduction section, consistent with the additions enumerated below. Compared to this earlier version [35] the present publication contains two additional sections:

- Section 4 contains an example of providing semantics for a language with exceptions

$$\tau ::= \mathsf{iProp} \mid 0 \mid \mathbf{1} \mid \mathbb{B} \mid \mathsf{Nat} \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau \mid \blacktriangleright \tau \mid I \mid \Sigma_{\mathtt{i} \in \mathtt{I}} \tau_{\mathtt{i}} \mid \Pi_{\mathtt{i} \in \mathtt{I}} \tau_{\mathtt{i}} \mid \ldots$$

$$t ::= x \mid F(t_1, \ldots, t_n) \mid \mathsf{abort}\ t \mid () \mid (t, t) \mid \pi_i\ t \mid \lambda x : \tau.\, t \mid$$
$$\mathsf{inj}_i\ t \mid \mathsf{match}\ t\ \mathsf{with}\ \overline{\mathsf{inj}_i\ x.t}\ \mathsf{end} \mid \mathsf{next}(t) \mid \mathit{fix}_\tau \mid \ldots$$

$$P ::= \mathsf{False} \mid t =_\tau t \mid P \vee P \mid P \to P \mid \forall x : \tau.\, P \mid P * P \mid P \mathrel{-\!\!*} P \mid \Box P \mid \boxed{P} \mid \triangleright P \mid \ldots$$

Fig. 1. Grammar for the Iris base logic.

- Section 7 contains details of extending Guarded Interaction Trees with support for preemptive concurrency with atomic effects

Sections 1 to 3, 5, 6 and 8 appear in the first author's PhD dissertation.

2 GUARDED INTERACTION TREES

In this section we provide an introduction to guarded interaction trees. Our treatment is brief, and we refer the reader to the original paper for details [18].

Iris and Guarded Type Theory. Guarded Interaction trees (GITrees) are defined in Iris logic. Here we briefly touch Iris, and refer the reader to the literature on Iris [20] and guarded type theory [9] for more in-depth details. Iris is a separation logic framework built on top of a model of *guarded type theory*, the main use of which is to solve recursive equations and define guarded recursive types, such as the type of GITrees described below. Moreover, Iris has a specialized proof mode [24], implemented in Rocq. This allows the users of Iris to carry out formal reasoning in separation logic as if they are proving things normally in Rocq, as we have done in the formalization of this work. For this reason, in the paper we will work with Iris and its type theory informally. Still, we need to say a few things about the foundations.

The syntax of Iris, shown in Figure 1, contains types, terms, and propositions. The grammar is standard for higher-order logic, with the exception of the guarded types fragment, and separation logic connectives. Here $\mathbb{B}$ is the type of booleans, $F(t_1, \ldots, t_n)$ denotes the application of a function symbol F from the logic's signature, and abort t is the eliminator of the empty type 0 (ex falso), producing a term of any type from $t : 0$. The type of propositions is denoted iProp. The *guarded* part of guarded type theory is the "later" modality $\blacktriangleright$. Intuitively, we view all types as indexed by a natural number, where τ_n contains elements of τ "at time" n. Then $\blacktriangleright\tau$ contains elements of τ at a later time; that is, $(\blacktriangleright\tau)_n = \tau_{n-1}$. There is an embedding $\mathsf{next} : \tau \to \blacktriangleright\tau$, and there is a *guarded* fixed point combinator $\mathit{fix}_\tau : (\blacktriangleright\tau \to \tau) \to \tau$, similar to the unguarded version in PCF. We can also lift functions to $\blacktriangleright$: given $f : A \to B$, we have $\blacktriangleright f : \blacktriangleright A \to \blacktriangleright B$.

For the proposition, Iris contains the usual separation logic connectives, and the two modalities: "later" $\triangleright$ and "persistently" $\Box$. The propositional $\triangleright$ modality reflects the type-level later modality $\blacktriangleright$ on the level of propositions, as justified by the following rule: $\triangleright(\alpha =_\tau \beta) \dashv\vdash \mathsf{next}(\alpha) =_{\blacktriangleright\tau} \mathsf{next}(\beta)$. The persistence modality $\Box P$ states that the proposition P is available without claiming any resources (as it normally is the case in separation logic); crucially it makes the proposition duplicable: $\Box P \vdash (\Box P) * (\Box P)$. An example of a persistent proposition is the invariant proposition $\boxed{P}$, which satisfies $\boxed{P} \vdash \Box\boxed{P}$.

Guarded Interaction Trees. Guarded recursive datatypes are datatypes obtained from recursive equations of the form $X = F(\blacktriangleright X)$. In other words, guarded recursive datatypes are similar to the regular datatypes you see in normal programming languages, but every recursive occurrence of

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

Fig. 2. Guarded datatype of interaction trees.

the type must be guarded by the $\blacktriangleright$ modality. The datatype we are concerned with here is the type of GITrees, shown in Figure 2. It is parameterized over two types: the ground type A and the effect signature E (more on it below).

Guarded Interaction Trees represent computational trees in which the leaves are of the ground type ($\mathsf{Ret}(a)$), error states ($\mathsf{Err}(e)$), and functions ($\mathsf{Fun}(f)$). The leaves $\mathsf{Ret}(a)$ and $\mathsf{Fun}(f)$ are also called *values*, and we write $\mathbf{IT}_E^v(A)$ for the type of $\mathbf{IT}_E(A)$-values. In the Rocq formalization, $\mathbf{IT}_E^v(A)$ is a separate type, connected to $\mathbf{IT}_E(A)$ by an injection $\mathit{IT_of_V} : \mathbf{IT}_E^v(A) \to \mathbf{IT}_E(A)$ and a partial projection $\mathit{IT_to_V} : \mathbf{IT}_E(A) \to \mathsf{option}\,\mathbf{IT}_E^v(A)$ that succeeds exactly on $\mathsf{Ret}(a)$ and $\mathsf{Fun}(g)$.

The nodes of the computation trees are of two kinds. The first one is a "silent step" constructor $\mathsf{Tau}(\alpha)$. It represents an unobservable internal step of the computation. For convenience, we use the function $\mathsf{Tick} \triangleq \mathsf{Tau} \circ \mathsf{next} : \mathbf{IT}_E(A) \to \mathbf{IT}_E(A)$ that "delays" its argument. This function satisfies the following equation: $\mathsf{Tick}(\alpha) = \mathsf{Tick}(\beta) \dashv\vdash \triangleright(\alpha = \beta)$.

The second kind of nodes are effects given by $\mathsf{Vis}_i(x, k)$. The parameters I, *Ins* and *Outs* are part of the effect signature E. The set I is the set of *names* of operations. The *arities* of an operation $i \in I$ are given by functors Ins_i and Outs_i. Let us give an example.

Consider the following signature for store effects. The signature E_{state} consists of effects $\{\mathtt{write}, \mathtt{read}, \mathtt{alloc}\}$ with the following input/output arities:

$$
\begin{aligned}
\mathit{Ins}_{\mathtt{write}}(X) &\triangleq \mathit{Loc} \times \blacktriangleright X & \mathit{Ins}_{\mathtt{read}}(X) &\triangleq \mathit{Loc} & \mathit{Ins}_{\mathtt{alloc}}(X) &\triangleq \blacktriangleright X \\
\mathit{Outs}_{\mathtt{write}}(X) &\triangleq \mathbf{1} & \mathit{Outs}_{\mathtt{read}}(X) &\triangleq \blacktriangleright X & \mathit{Outs}_{\mathtt{alloc}}(X) &\triangleq \mathit{Loc}
\end{aligned}
$$

For example, `write` expect a location and a new GITree as its input, and simply returns the unit value as an output. We write $\mathsf{Vis}_{\mathtt{write}}((\ell, \alpha), \lambda_.\, \beta)$ for the computation that invokes the `write` effect with arguments ℓ and α, waits for it to return, and proceeds as β. Thus, the first argument for Vis_i is the input, and the second one is the continuation dependent on the output. This continuation determines the branching in (G)Itrees.

For effects like above, it is usually convenient to provide wrappers:

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

When the signature and the return type are clear from the context, we simply write $\mathbf{IT}$ and $\mathbf{IT}^v$ for the GITrees and GITree-values.

Equational theory. GITrees come with a number of operations (defined using the recursion principle) that are used for writing and composing computations. Here we list some of those operations which we will be using. The function $\mathsf{get_val}(\alpha : \mathbf{IT}, f : \mathbf{IT}^v \to \mathbf{IT}) : \mathbf{IT}$ are used for sequencing computations, and its corresponding equations are shown in Figure 3. Intuitively,

$$\text{get_val} : \mathbf{IT}_E(A) \to (\mathbf{IT}_E^v(A) \to \mathbf{IT}_E(A)) \to \mathbf{IT}_E(A)$$

$$\begin{aligned}
\text{get_val}(\mathsf{Ret}(a), f) &= f(\mathsf{Ret}(a)) & \text{get_val}(\mathsf{Tau}(t), f) &= \mathsf{Tau}(\blacktriangleright \text{get_val}(t, f)) \\
\text{get_val}(\mathsf{Fun}(g), f) &= f(\mathsf{Fun}(g)) & \text{get_val}(\mathsf{Vis}_i(x, k), f) &= \mathsf{Vis}_i(x, \blacktriangleright \text{get_val}(-, f) \circ k) \\
\text{get_val}(\mathsf{Err}(e), f) &= \mathsf{Err}(e)
\end{aligned}$$

Fig. 3. Example function on Guarded Interaction Trees.

$$r : \prod_{\mathrm{i} \in E} \mathit{Ins}_{\mathrm{i}}(\mathbf{IT}_E) \times \mathit{State} \to \mathit{option}(\mathit{Outs}_{\mathrm{i}}(\mathbf{IT}_E) \times \mathit{State})$$

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

Fig. 4. Signature of reifiers and the reification function

$\text{get_val}(\alpha, f)$ first tries to compute α to a value (a $\mathsf{Ret}(a)$ or a $\mathsf{Fun}(g)$), and then calls f on that value. Similarly, $\text{get_fun}(\alpha : \mathbf{IT}, f : \blacktriangleright(\mathbf{IT} \to \mathbf{IT}) \to \mathbf{IT}) : \mathbf{IT}$ and $\text{get_ret}(\alpha : \mathbf{IT}_E(A), f : A \to \mathbf{IT}_E(A)) : \mathbf{IT}_E(A)$ first compute α to a value; if that value is a function $\mathsf{Fun}(g)$ (resp., $\mathsf{Ret}(a)$), then it proceeds with $f(g)$ (resp., $f(a)$). Otherwise it results in an runtime error.

Crucially, to work with higher-order computations, GITrees provide the "call-by-value" application $- \bullet - : \mathbf{IT}_E(A) \to \mathbf{IT}_E(A) \to \mathbf{IT}_E(A)$ satisfying the following equations:

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

where $-\ (\blacktriangleright\bullet)\ -$ is defined as the lifting of $- \bullet -$ to $\blacktriangleright\mathbf{IT}_E(A) \to \blacktriangleright\mathbf{IT}_E(A) \to \blacktriangleright\mathbf{IT}_E(A)$, and $\beta_v \in \mathbf{IT}_E^v(A)$ is either $\mathsf{Ret}(a)$ or $\mathsf{Fun}(g)$.

The application function $\alpha \bullet \beta$ simulates strict function application. It first tries to evaluate β to a value β_v. Then it tires to evaluate α to a function f. If it succeeds, then it invokes $f(\beta_v)$. If at any point it fails, application results in a runtime error.

For the often-used case of GITrees where the ground type includes natural numbers, we use the function $\mathsf{NatOp} : (\mathbb{N} \to \mathbb{N} \to \mathbb{N}) \to \mathit{IT} \to \mathit{IT} \to \mathit{IT}$ which lifts binary functions on natural numbers to binary functions on GITrees. That is, $\mathsf{NatOp}_f(\alpha, \beta)$ first evaluate GITrees β and α to values. If those values are natural numbers, then it computes f of those numbers and returns the result as a GITree. Otherwise, it returns a runtime error $\mathsf{Err}(\mathit{RunTime})$.

Reification and reduction relation. The semantics for effects are given in terms of *reifiers*. A reifier for the signature E is a tuple (State, r), where State is a type representing the internal state needed to reify the effects, and r is a reifier function of the type given in Figure 4. The idea is that r_i uses the internal state State to compute the output of the effect i based on its input.

For example, for the store effects we take *State* to be a map from locations to ▶**IT** (representing the heap); and we define the following reifier functions:

$$r_{\mathtt{write}}((\ell,\alpha),\sigma) = \mathsf{Some}((),\sigma[\ell \mapsto \alpha]) \qquad \text{(where } \ell \in \sigma\text{, and None otherwise)}$$
$$r_{\mathtt{read}}(\ell,\sigma) = \mathsf{Some}(\alpha,\sigma) \qquad \text{(where } \sigma(\ell) = \alpha\text{, and None otherwise)}$$
$$r_{\mathtt{alloc}}(\alpha,\sigma) = \mathsf{Some}(\ell,\sigma[\ell \mapsto \alpha]) \qquad \text{(where } \ell \notin \sigma\text{)}$$

Given reifiers for all the effects, we define a function reify : $\mathbf{IT} \times \mathit{State} \to \mathbf{IT} \times \mathit{State}$ (as in Figure 4) that, given (α,σ) reifies the top-level effect in α using the state σ, and returns the reified GITree and the updated state.

The reify function is then used to give reduction semantics for GITrees. We write $(\alpha,\sigma) \rightsquigarrow (\beta,\sigma')$ for such a reduction step. The definition of $\rightsquigarrow$ is given internally in the logic:

$$(\alpha,\sigma) \rightsquigarrow (\beta,\sigma') \triangleq \big(\alpha = \mathsf{Tick}(\beta) \wedge \sigma = \sigma'\big) \vee \big(\exists i\, x\, k.\, \alpha = \mathsf{Vis}_i(x,k) \wedge \mathsf{reify}(\alpha,\sigma) = (\mathsf{Tick}(\beta),\sigma')\big)$$

That is, either α is a "delayed" computation $\mathsf{Tick}(\beta)$, which then reduces to β; or it is an effect that can be reified. Recall that we write Tick for the composition Tau ∘ next.

Note that the reify function operates on the top-level effect of the GITree. But what if the top-level constructor is not Vis, *e.g.* if we have an effect inside an "evaluation context"? The role of evaluation contexts in GITrees is played by *homomorphisms*, which also allow us to bubble up necessary effects to the top of the GITree.

Definition 2.1 (Homomorphism). A map $f : \mathbf{IT} \to \mathbf{IT}$ is a homomorphism, written $f \in \mathit{Hom}$, if it satisfies:

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

For example, $\lambda x.\, \alpha \bullet x$ is a homomorphism, and so is $\lambda x.\, \mathsf{get_val}(x,f)$. On the other hand, $\lambda x.\, \mathsf{Vis}_{\mathtt{alloc}}(\mathsf{next}(x),k)$ (for some fixed k) is *not* a homomorphism.

Example. To illustrate programming with GITrees, we revisit the imperative factorial of Frumin et al. [18], written using the store effects:

$$\mathit{fact} \triangleq \lambda n.\ \mathsf{Alloc}(\mathsf{Ret}\ 1,\ \lambda \mathit{acc}.\ \mathsf{Alloc}(\mathsf{Ret}\ n,\ \lambda \ell.\ \mathit{body}\ ;\ \mathsf{Read}\ \mathit{acc})),$$
$$\mathit{body} \triangleq \mathsf{While}\ \mathsf{Read}\ \ell\ \mathsf{do}\ \mathsf{let}\ i = \mathsf{Read}\ \ell\ \mathsf{in}\ \mathsf{Write}(\mathit{acc}, \mathsf{NatOp}_{\times}(i, \mathsf{Read}\ \mathit{acc}))\ ;\ \mathsf{Write}(\ell, \mathsf{NatOp}_(i, \mathsf{Ret}\ 1))$$

Here the looping construct While, the sequencing (;), and the arithmetic operation NatOp are not primitive, but are themselves definable as operations on GITrees. Sequencing is $\alpha\ ;\ \beta \triangleq \mathsf{get_val}(\alpha, \lambda_.\ \beta)$, which evaluates α to a value, discards it, and proceeds with β; its evaluation rule is $v\ ;\ \beta \equiv \beta$ once the first argument has reduced to a value v. The loop is defined by guarded recursion is characterized by the unfolding equation:

$$\mathsf{While}\ \alpha\ \beta \equiv \mathsf{If}(\alpha,\ \beta\ ;\ \mathsf{Tick}(\mathsf{While}\ \alpha\ \beta),\ \mathsf{Ret}(()))$$

The loop evaluates the condition α and, if it is non-zero, runs the body β and loops again after a Tick that guards the recursive occurrence; otherwise it returns the unit value. Reasoning about *fact* then amounts to repeatedly unfolding While with this equation and discharging each sequencing combinator by its evaluation rule. It allocates an accumulator *acc* initialized to 1 and a counter ℓ initialized to n, then repeatedly multiplies *acc* by the current value of the counter and decrements the counter, until the counter reaches 0, at which point it returns *acc*. Whenever evaluation reaches a store operation, such as Read ℓ or Write$(\ell,-)$, the surrounding context, which consists of the

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

$$\textsc{wp-hom}\quad \frac{f \in \mathit{Hom}\quad \mathsf{wp}\ \alpha\ \{\beta_v.\ \mathsf{wp}\ f(\beta_v)\ \{\Phi\}\}}{\mathsf{wp}\ f(\alpha)\ \{\Phi\}}$$

Fig. 5. Selected weakest precondition rules.

homomorphisms arising from sequencing, NatOp, and the loop, is pushed into the continuation of the corresponding Vis node, bubbling the effect up to the top of the GITree. The effect is then reified, applying the store reifier and updating the heap state accordingly.

Program logic. In order to reason about GITrees, we employ the full power of the Iris separation logic framework. The program logic operates on the propositions of the form $\mathsf{wp}\ \alpha\ \{\Phi\}$. This weakest precondition proposition intuitively states that the GITree α is safe to reduce, and when it fully reduces, the resulting value satisfies the postcondition $\Phi : \mathbf{IT}^v \to \mathsf{iProp}$, a predicate over GITree-values. Another important predicate is $\mathsf{has_state}(\sigma)$, which signifies ownership of the current state σ.

In Figure 5 we show the rules, on which we focus in this work. Let us describe their meaning. The rule WP-REIFY allows us to symbolically execute effects in GITrees. It is given in a general form, and is used to derive domain-specific rules for concrete effects. Another important rule is WP-HOM which allows one to separate the reasoning about the computation from the reasoning about the context. The reason why WP-HOM is sound (this is going to be important in the next section when we make it unsound), is because the reduction $\rightsquigarrow$ of GITrees satisfies the following properties which allow one to disentangle a homomorphism from the GITree it's applied to:

Lemma 2.2. *Let f be a homomorphism. Then,*

- *$(\alpha,\sigma) \rightsquigarrow (\beta,\sigma')$ implies $(f(\alpha),\sigma) \rightsquigarrow (f(\beta),\sigma')$;*
- *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 $\triangleright(f(\beta) = \beta')$.*

Finally, as usual in Iris, the program logic satisfies an adequacy property, which allows one to relate propositions proved in the logic to the actual semantics:

Theorem 2.3. *Let α be an interaction tree and σ be a state such that*

$$\mathsf{has_state}(\sigma) \vdash \mathit{wp}\ \alpha\ \{\Phi\}$$

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 $\beta \neq \mathsf{Err}(e)$ for any error $e \in \mathtt{Error}$.

The role of meta-logic is played by the Rocq system; thus, the adequacy theorem allows us to relate proofs inside the program logic (Iris) to the proofs on the level of Rocq. This aspect is important in Iris and GITrees in general, but it is orthogonal to the work that we present in this paper. See [18] for more details.

3 CONTEXT-DEPENDENT REIFICATION

In this section we extend reification to handle context-dependent effects, using a language λ_{callcc} with `call/cc` as a concrete example. In Section 3.1 we present λ_{callcc}'s syntax and operational semantics (in the usual style with evaluation contexts). We then show why the current GITrees framework cannot be used as a denotational model for λ_{callcc} directly. In Section 3.2 we introduce our generalization of reification for context-dependent effects and corresponding extensions to the GITrees program logic. In Section 3.3 we demonstrate that our extension works as intended: we give a denotational semantics for λ_{callcc}, and we show how the general program logic for GITrees specializes to a logic for reasoning about `call/cc`. We prove soundness and computational adequacy of denotational semantics using a logical relation defined within our program logic.

3.1 Operational Semantics and Type System for λ_{callcc}

By λ_{callcc} we denote a simply-typed λ-calculus with natural numbers, recursive functions, and `call/cc`. The relevant pieces of syntax, the type system and the operational semantics are given in Figure 6. The type system includes natural numbers, function types, and the type $\mathit{cont}(\tau)$ of continuations. The call/cc (x. e) expression takes the current evaluation context and binds it to x in e. The throw e to e' expression evaluates passes the first argument (value) to the second argument (continuation, represented as an evaluation context).

The operational semantics of λ_{callcc} is separated into two layers. The first layer consists of local reductions of primitive expressions ($e \mapsto_K e'$), and the second layer lifts local reductions to reductions among complete programs ($K[e] \mapsto K'[e']$). The local reductions are parameterized by an evaluation context K, which allows call/cc (x. e) to capture the evaluation context. Let us now consider what happens if we try to give a direct-style denotational semantics of λ_{callcc} into GITrees. By direct we mean that we wish to give a direct interpretation of types and expressions, rather than going through a global CPS conversion. To define the semantics, we first need to provide an effect signature, state, and reifiers for each effect, and then we can define the interpretation of the expressions of the language.

The effect signature, shown in Figure 7, contains two effects `callcc` and `throw`. Since `call/cc` binds the current continuation in its body, it is natural to let the input arity of `callcc` be a callback $(\blacktriangleright\mathbf{IT} \to \blacktriangleright\mathbf{IT}) \to \blacktriangleright\mathbf{IT}$: the callback is the body of `call/cc`, and its argument $\blacktriangleright\mathbf{IT} \to \blacktriangleright\mathbf{IT}$ is the captured continuation, represented as a (guarded) function that resumes the computation with a supplied value. The output arity $\blacktriangleright\mathbf{IT}$ is the computation that `call/cc` eventually returns.

The input arity for `throw` signifies that `throw` takes as input an expression and a continuation, which are represented respectively as $\blacktriangleright\mathbf{IT}$ and $\blacktriangleright(\mathbf{IT} \to \mathbf{IT})$. The output arity of `throw` is simply the empty type 0, because `throw` never returns.

Note that the input types of `callcc` and `throw` have slightly different arities. However, we can always transform $f : (\blacktriangleright X \to \blacktriangleright X)$ into an element of type $\blacktriangleright(X \to X)$ by performing a silent step in the function's body: $f' \triangleq \mathsf{next}(\lambda x.\, \mathsf{Tau}(f(\mathsf{next}(x))))$. And we can always transform $f : \blacktriangleright(X \to X)$ into an element of type $\blacktriangleright X \to \blacktriangleright X$ using the applicative structure of the later modality.

For convenience, we will use the abbreviations $\mathrm{Callcc}(f)$ and $\mathrm{Throw}(e)$, defined in Figure 7, for representing denotations of `throw` and `call/cc` as effects in GITrees.

To complete all the ingredients for the denotational semantics, we need reifiers for the `callcc` and `throw` effects. Given our operational understanding of continuations, the natural choice for the local state type *State* is **1** (since we do not have any state). However, the current reifier signature (Figure 4) poses a problem. Reifiers, as they are now, cannot access their current continuation, which is essential for both effects. $\mathrm{Callcc}(f)$ needs to pass the current continuation to f, while Throw must redirect control to a provided continuation instead of returning normally. The current

$$
\begin{array}{lll}
\text{types} & Ty \ni \tau & ::= \mathbb{N} \mid \tau_1 \to \tau_2 \mid cont(\tau) \\
\text{expressions} & Expr \ni e & ::= v \mid x \in Var \mid e_1\ e_2 \mid e_1 \oplus e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
 & & \quad \mid \text{call/cc } (\mathsf{x}.\ e) \mid \text{throw } e_1 \text{ to } e_2 \\
\text{values} & Val \ni v & ::= n \mid \text{rec } f(x) = e \mid \text{cont } K \\
\text{eval. cont.} & Ectx \ni K & ::= \square \mid \text{if } K \text{ then } e_1 \text{ else } e_2 \mid K\ v \mid e\ K \mid e \oplus K \mid K \oplus v \\
 & & \quad \mid \text{throw } K \text{ to } e \mid \text{throw } v \text{ to } K
\end{array}
$$

$$
(\text{rec } f(x) = e)\ v \mapsto_K e[\text{rec } f(x) = e/f][v/x] \qquad \frac{n = 0}{\text{if } n \text{ then } e_1 \text{ else } e_2 \mapsto_K e_2}
$$

$$
\frac{0 < n}{\text{if } n \text{ then } e_1 \text{ else } e_2 \mapsto_K e_1} \qquad \frac{n_1 \oplus n_2 = n_3}{n_1 \oplus n_2 \mapsto_K n_3} \qquad \text{call/cc } (\mathsf{x}.\ e) \mapsto_K e[\text{cont } K/x]
$$

$$
K[\text{throw } v \text{ to cont } K'] \mapsto K'[v] \qquad \frac{e_1 \mapsto_K e_2}{K[e_1] \mapsto K[e_2]}
$$

$$
\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash n : \mathbb{N}} \qquad \frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{rec } f(x) = e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}
$$

$$
\frac{\Gamma \vdash e_1 : \mathbb{N} \qquad \Gamma \vdash e_2 : \mathbb{N}}{\Gamma \vdash e_1 \oplus e_2 : \mathbb{N}} \qquad \frac{\Gamma \vdash e_1 : \mathbb{N} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \qquad \frac{\Gamma, x : cont(\tau) \vdash e : \tau}{\Gamma \vdash \text{call/cc } (\mathsf{x}.\ e) : \tau}
$$

$$
\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : cont(\tau)}{\Gamma \vdash \text{throw } e_1 \text{ to } e_2 : \tau'}
$$

Fig. 6. Syntax, type system and operational semantics of λ_{callcc}.

$$
\begin{array}{ll}
Ins_{\texttt{callcc}}(X) \triangleq ((\blacktriangleright X \to \blacktriangleright X) \to \blacktriangleright X) & Outs_{\texttt{callcc}}(X) \triangleq \blacktriangleright X \\
Ins_{\texttt{throw}}(X) \triangleq \blacktriangleright X \times \blacktriangleright (X \to X) & Outs_{\texttt{throw}}(X) \triangleq 0 \\
\text{Callcc}(f) \triangleq \text{Vis}_{\texttt{callcc}}(f, \text{id}) & \text{Throw}(e, f) \triangleq \text{Vis}_{\texttt{throw}}(e, f, \lambda x.\ \text{abort } x)
\end{array}
$$

Fig. 7. Signatures and opertaions on GITrees with `call/cc`.

reifiers lacks this capability, and in the next subsection we show how to generalize the notion of reification to context-dependent effects.

3.2 Context-dependent Reifiers

This section presents our extension to context-dependent reification, and the limitations it imposes on the program logic. In order to allow reifiers to manage continuations, we change the type of reifiers to accept continuations as an extra parameter, as shown in Figure 8. Continuations for a given effect are functions from the effect's outputs to GITrees: $Outs_i(\mathbf{IT}_E) \to \blacktriangleright \mathbf{IT}_E$. Given a set of context-dependent reifiers, we define a context-dependent `reify` function, also shown in Figure 8. As before, `reify` dispatches to the correct individual reifier for the effect. Note that now it is the user's responsibility to pass the output of an effect to the given continuation if the control flow

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

$$\frac{r_i(x, \sigma, \kappa) = \mathsf{Some}(\beta, \sigma')}{\mathsf{reify}(\mathsf{Vis}_i(x, \kappa), \sigma) = (\mathsf{Tau}(\beta), \sigma')} \qquad \frac{r_i(x, \sigma, \kappa) = \mathsf{None}}{\mathsf{reify}(\mathsf{Vis}_i(x, \kappa), \sigma) = (\mathsf{Err}(\mathit{RunTime}), \sigma)}$$

Fig. 8. Type of context-dependent reifiers and the context-dependent reify function

WP-WRITE

$$\frac{\mathsf{heap_ctx} \quad \triangleright \ell \mapsto \alpha \quad \triangleright(\ell \mapsto \beta \twoheadrightarrow \mathsf{wp}\ \mathsf{Ret}()\ \{\Phi\})}{\mathsf{wp}\ \mathsf{Write}(\ell, \beta)\ \{\Phi\}}$$

WP-WRITE-CTX-DEP

$$\frac{\kappa \in \mathit{Hom} \quad \mathsf{heap_ctx} \quad \triangleright \ell \mapsto \alpha \quad \triangleright(\ell \mapsto \beta \twoheadrightarrow \mathsf{wp}\ (\kappa\ (\mathsf{Ret}()))\ \{\Phi\})}{\mathsf{wp}\ (\kappa\ (\mathsf{Write}(\ell, \beta)))\ \{\Phi\}}$$

WP-REIFY-CTX-DEP

$$\frac{\mathsf{has_state}(\sigma) \quad r_i(x, \sigma, k) = \mathsf{Some}(\mathsf{next}(\beta), \sigma') \quad \triangleright\ (\mathsf{has_state}(\sigma') \twoheadrightarrow \mathsf{wp}\ \beta\ \{\Phi\})}{\mathsf{wp}\ \mathsf{Vis}_i(x, k)\ \{\Phi\}}$$

Fig. 9. Program logic in the presence of context-dependent reifiers.

is not supposed to be interrupted. For example, since the evaluation of a call/cc (x. e) expression does not modify the control flow itself, but simply passes the current continuation to its body, the context-dependent reifier for `callcc` is simply $r_{\mathtt{callcc}}(x, \sigma, \kappa) = \mathsf{Some}(\kappa\ (x\ \kappa), \sigma)$.

Before we move on to discussing the consequences of this for program logic, we would like to note that our treatment of continuation (with top-level reifiers dispatching them) parallels Cartwright and Felleisen's "extensible direct models" [11], which also aimed to support extensible denotational semantics in classical domain theory. We discuss this more in Section 8.

Program logic for GITrees in the presence of context-dependent reifiers. To reflect the generalization to context-dependent reifiers in the program logic, we replace the proof rule WP-REIFY by WP-REIFY-CTX-DEP, shown in Figure 9. This is, however, not the only change we need to make. In the presence of context-dependent effects, WP-HOM is not sound! (A similar observation was also made by [36] in their development of a program logic for `call/cc`.) The reason is that context-dependent reification invalidates Lemma 2.2. Now, since WP-HOM is no longer sound, one might worry that *every* rule must be reformulated to thread an explicit homomorphism, just as Timany and Birkedal [36] thread an explicit evaluation context through their rules. Fortunately, only the rules for effectful operations need such a change. The reason is that our program logic reasons about *denotations*, whose equational theory lets us manipulate the effect-free part of a GITree directly, without ever mentioning the ambient homomorphism. For instance, WP-WRITE from [18] is strengthened to WP-WRITE-CTX-DEP, and considers ambient homomorphisms explicitly. It is possible, because Write effect does not need to inspect the evaluation context. The premise heap_ctx in these rules is a persistent invariant asserting the existence of a heap: it connects the points-to assertions $\ell \mapsto v$ (ghost fragments recording that location ℓ holds v) to the underlying state ownership $\mathsf{has_state}(\sigma)$. Being an Iris invariant, heap_ctx is freely duplicable and can thus be shared throughout the program (and, in Section 7, between threads); a client opens it to access the heap when performing a heap operation. We give its formal definition in Section 7.5.

Our context-dependent reification extension, though simple, allows us to build sound and adequate denotational models for languages with control-flow operators, including $\lambda_{\mathtt{callcc}}$ (shown in the next subsection). Moreover, our extension is conservative, and we recover previous case studies

$$\mathbf{E}[\![x]\!]_\rho = \rho(x)$$
$$\mathbf{E}[\![v]\!]_\rho = \mathbf{V}[\![v]\!]_\rho$$
$$\mathbf{E}[\![e_1\ e_2]\!]_\rho = \mathbf{E}[\![e_1]\!]_\rho \bullet \mathbf{E}[\![e_2]\!]_\rho$$
$$\mathbf{E}[\![e_1 \oplus e_2]\!]_\rho = \mathsf{NatOp}_\oplus(\mathbf{E}[\![e_1]\!]_\rho, \mathbf{E}[\![e_2]\!]_\rho)$$
$$\mathbf{E}[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!]_\rho = \mathsf{If}(\mathbf{E}[\![e_1]\!]_\rho, \mathbf{E}[\![e_2]\!]_\rho, \mathbf{E}[\![e_3]\!]_\rho)$$
$$\mathbf{E}[\![\mathsf{call/cc}\ (\mathrm{x}.\ e)]\!]_\rho = \mathsf{Callcc}(\lambda(f : \blacktriangleright\mathbf{IT} \to \blacktriangleright\mathbf{IT}).\ \mathbf{E}[\![e]\!]_{\rho[x \mapsto \mathsf{Fun}(\mathsf{next}(\lambda y.\ \mathsf{Tau}(f(\mathsf{next}(y)))))]})$$
$$\mathbf{E}[\![\mathsf{throw}\ e_1\ \mathsf{to}\ e_2]\!]_\rho = \mathsf{get_val}(\mathbf{E}[\![e_1]\!]_\rho, \lambda x.\ \mathsf{get_fun}(\mathbf{E}[\![e_2]\!]_\rho, \lambda f.\ \mathsf{Throw}(x, f)))$$
$$\mathbf{V}[\![\mathsf{cont}\ K]\!]_\rho = \mathsf{Fun}(\mathsf{next}(\lambda x.\ \mathsf{Tau}(\mathbf{K}[\![K]\!]_\rho\ (\blacktriangleright\bullet)\ \mathsf{next}(x))))$$
$$\mathbf{V}[\![n]\!]_\rho = \mathsf{Ret}(n)$$
$$\mathbf{V}[\![\mathsf{rec}\ f(x) = e]\!]_\rho = \mathsf{Fun}(\mathsf{next}(\lambda a.\ \mathbf{E}[\![e]\!]_{\rho[f \mapsto \mathbf{V}[\![\mathsf{rec}\ f(x)=e]\!]_\rho,\ x \mapsto a]}))$$
$$\mathbf{K}[\![\square]\!]_\rho = \lambda x.\ x$$
$$\mathbf{K}[\![e\ K]\!]_\rho = \lambda x.\ \mathbf{E}[\![e]\!]_\rho \bullet (\mathbf{K}[\![K]\!]_\rho\ x)$$
$$\mathbf{K}[\![K\ v]\!]_\rho = \lambda x.\ (\mathbf{K}[\![K]\!]_\rho\ x) \bullet \mathbf{V}[\![v]\!]_\rho$$
$$\mathbf{K}[\![e \oplus K]\!]_\rho = \lambda x.\ \mathsf{NatOp}_\oplus(\mathbf{E}[\![e]\!]_\rho, \mathbf{K}[\![K]\!]_\rho\ x)$$
$$\mathbf{K}[\![K \oplus v]\!]_\rho = \lambda x.\ \mathsf{NatOp}_\oplus(\mathbf{K}[\![K]\!]_\rho\ x, \mathbf{V}[\![v]\!]_\rho)$$
$$\mathbf{K}[\![\mathsf{if}\ K\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]_\rho = \lambda x.\ \mathsf{If}(\mathbf{K}[\![K]\!]_\rho\ x, \mathbf{E}[\![e_1]\!]_\rho, \mathbf{E}[\![e_2]\!]_\rho)$$
$$\mathbf{K}[\![\mathsf{throw}\ K\ \mathsf{to}\ e]\!]_\rho = \lambda x.\ \mathsf{get_val}(\mathbf{K}[\![K]\!]_\rho\ x, \lambda y.\ \mathsf{get_fun}(\mathbf{E}[\![e]\!]_\rho, \lambda f.\ \mathsf{Throw}(y, f)))$$
$$\mathbf{K}[\![\mathsf{throw}\ v\ \mathsf{to}\ K]\!]_\rho = \lambda x.\ \mathsf{get_val}(\mathbf{V}[\![v]\!]_\rho, \lambda y.\ \mathsf{get_fun}(\mathbf{K}[\![K]\!]_\rho\ x, \lambda f.\ \mathsf{Throw}(y, f)))$$

Fig. 10. Denotational semantics of $\lambda_{\mathsf{callcc}}$.

(computational adequacy of $\lambda_{\mathsf{rec,io}}$ and type safety for $\lambda_{\multimap,\mathsf{ref}}$ [18]) with minimal modifications; see the accompanying Rocq formalization.

3.3 Denotational Semantics of $\lambda_{\mathsf{callcc}}$

In this section we show that context-dependent reifiers are sufficient for providing a sound and adequate semantic model of $\lambda_{\mathsf{callcc}}$. We define context-dependent reifiers for `callcc` and `throw`, then prove that this gives a sound interpretation w.r.t. operational semantics. To show adequacy, we define a logical relation, which relates the denotational and operational semantics. The logical relation is defined in the (updated) program logic for GITrees (following the approach in [18]), and validates the utility of the program logic.

Interpretation of $\lambda_{\mathsf{callcc}}$. The denotational semantics of $\lambda_{\mathsf{callcc}}$ is shown in Figure 10. The interpretation is split into three parts: $\mathbf{E}[\![-]\!]$ for expressions, $\mathbf{V}[\![-]\!]$ for values, and $\mathbf{K}[\![-]\!]$ for evaluation contexts. Throughout, the functions get_val and get_fun (cf. Figure 3) pin down the left-to-right evaluation order: they first evaluate their argument to a GITree value, a function value in the case of get_fun, and then pass it on. In the clause for $\mathsf{throw}\ e_1\ \mathsf{to}\ e_2$, for example, they force e_1 and then e_2 before the effect Throw fires.

The last two clauses give the denotations of the two evaluation contexts for `throw` (cf. Figure 6), namely $\mathsf{throw}\ K\ \mathsf{to}\ e$ and $\mathsf{throw}\ v\ \mathsf{to}\ K$, which place the hole in the value position and in the continuation position, respectively. The denotation of a context is a function $\lambda x.\ \ldots$ of the GITree x that fills the hole: it plugs x into the surrounding context via $\mathbf{K}[\![K]\!]_\rho\ x$, evaluates the remaining

subterm, and finally performs Throw. The same use of get_val and get_fun fixes the evaluation order, so that throw K to e forces $K[x]$ first, whereas throw v to K forces the already-computed value v before $K[x]$, and both clauses agree with the throw e_1 to e_2 expression clause once the hole is filled.

The context-dependent reifiers for the effects `callcc` and `throw` are defined as follows:

$$r_{\texttt{callcc}}(f, (), \kappa) = \mathsf{Some}(\kappa\ (f\ \kappa), ()) \qquad r_{\texttt{throw}}((\alpha, f), (), \kappa) = \mathsf{Some}(f\ \alpha, ())$$

To show that the denotational semantics is sound, we need the following lemma that shows that interpretations of expressions in evaluation contexts are decomposed into applications of homomorphisms.

Lemma 3.1. *For any context K and an environment ρ, we have $\mathbf{K}[\![K]\!]_\rho \in \mathit{Hom}$. For any context K, expression e, and an environment ρ, $\mathbf{E}[\![K[e]]\!]_\rho = \mathbf{K}[\![K]\!]_\rho(\mathbf{E}[\![e]\!]_\rho)$.*

With these results at hand, we can show soundness of our interpretation:

Lemma 3.2. *Soundness. Suppose $e_1 \mapsto e_2$. Then $(\mathbf{E}[\![e_1]\!]_\rho, ()) \rightsquigarrow^* (\mathbf{E}[\![e_2]\!]_\rho, ())$, where $() : \mathbf{1}$ is the unique element of the unit type, representing the (lack of) state.*

Program logic for $\lambda_{\mathsf{callcc}}$. We now specialize the general program logic rule WP-REIFY-CTX-DEP using the reifiers for `callcc` and `throw` to obtain the following program logic rules:

WP-THROW

$$\frac{\kappa \in \mathit{Hom} \quad \mathsf{has_state}(\sigma) \quad \triangleright(\mathsf{has_state}(\sigma) \mathrel{-\!\!*} \mathsf{wp}\ (f\ x)\ \{\Phi\})}{\mathsf{wp}\ (\kappa\ (\mathsf{Throw}(\mathsf{next}(x), \mathsf{next}(f))))\ \{\Phi\}}$$

WP-CALLCC

$$\frac{\kappa \in \mathit{Hom} \quad \mathsf{has_state}(\sigma) \quad \triangleright(\mathsf{has_state}(\sigma) \mathrel{-\!\!*} \mathsf{wp}\ (\kappa\ (f\ \kappa))\ \{\Phi\})}{\mathsf{wp}\ (\kappa\ (\mathsf{Callcc}(\mathsf{next} \circ f)))\ \{\Phi\}}$$

where κ is a homomorphism representing the current evaluation context on the level of GITrees. The reader may wonder why these rules include the $\mathsf{has_state}(\sigma)$ predicates, since it is just 'threaded around'. The reason is that these rules also apply when there are other effects around and the state is composed of different substates for different effects, cf. the discussion of modularity in Section 2.

Adequacy and logical relation. Having established soundness, we now turn our attention to *adequacy*, which is usually much more complicated to prove.

Lemma 3.3. *Adequacy. Suppose that $\emptyset \vdash e : \mathbb{N}$ and $(\mathbf{E}[\![e]\!]_\emptyset, \sigma_1) \rightsquigarrow^* (\mathsf{Ret}(n), \sigma_2)$, for a natural number n. Then $e \mapsto^* n$.*

To prove Lemma 3.3, we define a logical relation between syntax ($\lambda_{\mathsf{callcc}}$ programs) and semantics (GITrees denotations) using the program logic from Figure 11. To handle control effects, we use a *biorthogonal* logical relation [30], adapted from [36] for adequacy, following the Iris approach [38].

The core observational refinement $\mathcal{O}(\alpha, e)$ ensures that if α reduces to a GITree value $\mathbf{IT}^v$, then this value is a natural number, and e also reduces to the same number. The evaluation context relation $\mathcal{K}(P)(\kappa, K)$ relates homomorphisms and evaluation contexts when they map related arguments to expressions satisfying $\mathcal{O}$. The expression relation $\mathcal{E}(P)(\alpha, e)$ connects related $\mathbf{IT}$'s and expressions in related evaluation contexts. Types are inductively interpreted: functions relate if they map related arguments to related results, and continuations relate via the context relation. For open terms, the validity judgment $\Gamma \vDash e : \tau$ uses closing substitutions, with $e[\gamma]$ denoting applying a substitution γ to e.

The proof of adequacy relies on the fact that the interpretation of evaluation contexts are homomorphisms, which allows us to use a limited version of the bind rule:

Lemma 3.4. *Limited bind rule. If $\mathcal{E}(P)(\alpha, e)$ and $\mathcal{K}(P)(\kappa, K)$, then $\mathcal{O}(\kappa\ \alpha, K[e])$.*

$$\mathcal{O}(\alpha, e) \triangleq \mathsf{has_state}(()) \twoheadrightarrow \mathsf{wp}\ \alpha\ \{\beta.\exists v.\ (e \mapsto^* v) * [\![\mathbb{N}]\!](\beta, v) * \mathsf{has_state}(())\} \qquad \boxed{\mathcal{O} : \mathrm{ERel}}$$

$$\mathcal{K}(R)(\kappa, K) \triangleq \square\, \forall(\beta, v).\ R(\beta, v) \twoheadrightarrow \mathcal{O}(\kappa\ \beta, K[v]) \qquad \boxed{\mathcal{K} : \mathrm{VRel} \to \mathrm{CRel}}$$

$$\mathcal{E}(R)(\alpha, e) \triangleq \forall(\kappa, K).\ \mathcal{K}(R)(\kappa, K) \twoheadrightarrow \mathcal{O}(\kappa\ \alpha, K[e]) \qquad \boxed{\mathcal{E} : \mathrm{VRel} \to \mathrm{ERel}}$$

$$[\![\mathbb{N}]\!](\alpha, v) \triangleq \exists n : \mathbb{N}.\ \alpha = \mathsf{Ret}(n) \wedge v = n \qquad \boxed{[\![\tau]\!] : \mathrm{VRel}}$$

$$[\![\tau_1 \to \tau_2]\!](\beta, v) \triangleq \exists f.\ \beta = \mathsf{Fun}(f) \wedge \square\, \forall(\alpha, v').\ [\![\tau_1]\!](\alpha, v') \twoheadrightarrow \mathcal{E}([\![\tau_2]\!])(\mathsf{Fun}(f) \bullet \alpha, v\ v')$$

$$[\![\mathit{cont}(\tau)]\!](\beta, v) \triangleq \exists \kappa\ K.\ \beta = \mathsf{Fun}(\mathsf{next}(\lambda x.\ \mathsf{Tick}(\kappa\ x))) \wedge v = \mathsf{cont}\ K \wedge \mathcal{K}([\![\tau]\!])(\kappa, K)$$

$$[\![\Gamma]\!](\rho, \gamma) \triangleq \forall(x : \tau) \in \Gamma.\ [\![\tau]\!](\rho\ x, \gamma\ x)$$

$$\Gamma \vDash e : \tau \triangleq \square\, \forall(\rho, \gamma).\ [\![\Gamma]\!](\rho, \gamma) \twoheadrightarrow \mathcal{E}([\![\tau]\!])(\mathbf{E}[\![e]\!]_\rho, e[\gamma])$$

$$\boxed{\begin{array}{l} \mathrm{CRel} \triangleq \mathit{Hom} \times \mathit{Ectx} \to \mathit{iProp} \\ \mathrm{VRel} \triangleq \mathbf{IT}^v \times \mathit{Val} \to \mathit{iProp} \\ \mathrm{ERel} \triangleq \mathbf{IT} \times \mathit{Expr} \to \mathit{iProp} \end{array}}$$

Fig. 11. Logical relation for $\lambda_{\mathsf{callcc}}$.

With this in mind we show the fundamental lemma, stating that every well-typed expression is related to its own interpretation:

Lemma 3.5. *Fundamental lemma. Let* $\Gamma \vdash e : \tau$ *then* $\Gamma \vDash e : \tau$.

Computational adequacy now follows easily from the fundamental lemma.

Proof of Lemma 3.3. By Lemma 3.5, we have that $\emptyset \vdash e : \mathbb{N}$ implies that $\emptyset \vDash e : \mathbb{N}$. Now, the statement follows from Theorem 2.3 and the assumption that $\mathbf{E}[\![e]\!]_\emptyset, \sigma_1 \rightsquigarrow^* \mathsf{Ret}\ n, \sigma_2$. □

4 EXCEPTIONS

Having seen how to model undelimited continuations (`call/cc` and `throw`) in Section 3, we now turn to a more structured form of control flow: exceptions. Exceptions represent an important middle ground between the full generality of `call/cc` and the more restricted delimited continuations we will study in Section 5. While `call/cc` allows capturing and invoking arbitrary continuations, exceptions provide a more disciplined mechanism for non-local control transfer, where handlers are explicitly registered and exceptions can only jump to their nearest enclosing handler.

This section serves two purposes. First, it demonstrates that our framework naturally handles exceptions, which are ubiquitous in mainstream programming languages. Second, and more importantly, it provides a *gentle introduction* to using context-dependent effects in GITrees by walking through the design decisions involved in modeling exceptions. We show how to decompose the operational behavior of exceptions into primitive effects, how to maintain appropriate state, and how the program logic rules follow naturally from the effect signatures.

The key insight is that exceptions require managing a *stack of handlers*. Unlike `call/cc`, which captures the entire continuation as a first-class value, exceptions maintain a stack of handler contexts that must be consulted when an exception is thrown.

We proceed as follows. In Section 4.1 we present $\lambda_{\mathsf{rec,exc}}$, a lambda calculus with exceptions. In Section 4.2 we show how to model exceptions in GITrees, explaining the design choices step by step. We conclude by deriving program logic rules for reasoning about exceptions.

$$
\begin{array}{lll}
\text{expressions} & \mathit{Expr} \ni e & ::= v \mid x \in \mathit{Var} \mid e_1\ e_2 \mid e_1 \oplus e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
& & \mid \text{try } e_1 \text{ catch } \mathit{err} \text{ with } (\mathsf{h}.\ e_2) \mid \text{throw } \mathit{err}\ e \\
\text{values} & \mathit{Val} \ni v & ::= n \mid \text{rec } f(x) = e \\
\text{eval. contexts} & \mathit{Ectx} \ni K & ::= \square \mid K[\square\ v] \mid K[e\ \square] \mid K[e \oplus \square] \mid K[\square \oplus v] \mid K[\text{if } \square \text{ then } e_1 \text{ else } e_2] \\
& & \mid K[\text{try } \square \text{ catch } \mathit{err} \text{ with } (\mathsf{h}.\ e)] \mid K[\text{throw } \mathit{err}\ \square]
\end{array}
$$

Fig. 12. Syntax of $\lambda_{\text{rec,exc}}$.

4.1 Lambda calculus with first-order exceptions, $\lambda_{\text{rec,exc}}$

We begin by defining the language we wish to model. The language $\lambda_{\text{rec,exc}}$, shown in Figure 12, extends the simply-typed lambda calculus with recursive functions with two constructs for exception handling:

- $\text{throw } \mathit{err}\ e$ raises an exception named err with payload e;
- $\text{try } e_1 \text{ catch } \mathit{err} \text{ with } (\mathsf{h}.\ e_2)$ evaluates e_1 in a context where exceptions named err are caught by the handler $h.\ e_2$.

We fix a countable set *Exc* of exception names with decidable equality. Exception names allow scoping: a `catch` block only handles exceptions with a matching name, and different exception names can be handled by different handlers. This is similar to typed exceptions in languages like Java or OCaml, where different exception types can be caught separately.

Operational Semantics. The operational semantics, shown in Figure 13, uses a CEK-style abstract machine with four configurations:

- $\langle e \rangle_{\text{term}}$: initial configuration, about to execute the term e;
- $\langle e,\ K \rangle_{\text{eval}}$: evaluation mode, decomposing e and building continuation stack K;
- $\langle K,\ v \rangle_{\text{cont}}$: continuation mode, processing the stack K with value v;
- $\langle v \rangle_{\text{ret}}$: final configuration, returning value v.

The key rules for exceptions are:

(1) When evaluating $\text{try } e_1 \text{ catch } \mathit{err} \text{ with } (\mathsf{h}.\ e_2)$, we push a handler evaluation context $\text{try } \square \text{ catch } \mathit{err} \text{ with } (\mathsf{h}.\ e_2)$ onto the stack and proceed to evaluate e_1

(2) If e_1 evaluates to a value without raising an exception, the handler evaluation context is simply discarded:

$$\langle K[\text{try } \square \text{ catch } \mathit{err} \text{ with } (\mathsf{h}.\ e)],\ v \rangle_{\text{cont}} \mapsto \langle v,\ K \rangle_{\text{eval}}$$

(3) If an exception err is thrown, we search the continuation stack for the *nearest* matching handler, discarding all intermediate evaluation contexts:

$$\langle K'[\text{try } K[\text{throw } \mathit{err}\ \square] \text{ catch } \mathit{err} \text{ with } (\mathsf{h}.\ e)],\ v \rangle_{\text{cont}} \mapsto \langle e[v/\mathsf{h}],\ K' \rangle_{\text{eval}}$$

where K contains no handler for err.

This operational semantics captures the essential behavior of exceptions: handlers are installed on a stack, and throwing an exception unwinds the stack to the nearest matching handler.

Example. Consider the program $C \triangleq \text{try } \big(\text{throw } \mathit{err}_1\ (\text{throw } \mathit{err}_2\ 5)\big) \text{ catch } \mathit{err}_2 \text{ with } (\mathsf{h}.\ \mathsf{h} \oplus 1)$, whose handler increments the value it catches. In the abstract machine of Figure 13 it evaluates as

abstract machine config. $\mathit{Config} ::= \langle e,\ K\rangle_{\mathsf{eval}} \mid \langle K,\ v\rangle_{\mathsf{cont}} \mid \langle e\rangle_{\mathsf{term}} \mid \langle v\rangle_{\mathsf{ret}}$

$$\begin{aligned}
\langle e\rangle_{\mathsf{term}} &\mapsto \langle e,\ \square\rangle_{\mathsf{eval}} \\
\langle \square,\ v\rangle_{\mathsf{cont}} &\mapsto \langle v\rangle_{\mathsf{ret}} \\
\langle e_0\ e_1,\ K\rangle_{\mathsf{eval}} &\mapsto \langle e_1,\ K[e_0\ \square]\rangle_{\mathsf{eval}} \\
\langle v,\ K\rangle_{\mathsf{eval}} &\mapsto \langle K,\ v\rangle_{\mathsf{cont}} \\
\langle K[e\ \square],\ v\rangle_{\mathsf{cont}} &\mapsto \langle e,\ K[\square\ v]\rangle_{\mathsf{eval}} \\
\langle K[(\mathsf{rec}\, f(x) = e)\ \square],\ v\rangle_{\mathsf{cont}} &\mapsto \langle e[v/x][\mathsf{rec}\, f(x) = e/f],\ K\rangle_{\mathsf{eval}} \\
\langle e_0 \oplus e_1,\ K\rangle_{\mathsf{eval}} &\mapsto \langle e_1,\ K[e_0 \oplus \square]\rangle_{\mathsf{eval}} \\
\langle K[e_0 \oplus \square],\ v\rangle_{\mathsf{cont}} &\mapsto \langle e_0,\ K[\square \oplus v]\rangle_{\mathsf{eval}} \\
\langle K[\square \oplus v_2],\ v_1\rangle_{\mathsf{cont}} &\mapsto \langle v_3,\ K\rangle_{\mathsf{eval}} \quad (v_1 \oplus v_2 = v_3) \\
\langle \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3,\ K\rangle_{\mathsf{eval}} &\mapsto \langle e_1,\ K[\mathsf{if}\ \square\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\rangle_{\mathsf{eval}} \\
\langle K[\mathsf{if}\ \square\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3],\ n\rangle_{\mathsf{cont}} &\mapsto \langle e_3,\ K\rangle_{\mathsf{eval}} \quad (n = 0) \\
\langle K[\mathsf{if}\ \square\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3],\ n\rangle_{\mathsf{cont}} &\mapsto \langle e_2,\ K\rangle_{\mathsf{eval}} \quad (0 < n) \\
\langle \mathsf{try}\ e_1\ \mathsf{catch}\ \mathit{err}\ \mathsf{with}\ (\mathsf{h}.\ e_2),\ K\rangle_{\mathsf{eval}} &\mapsto \langle e_1,\ K[\mathsf{try}\ \square\ \mathsf{catch}\ \mathit{err}\ \mathsf{with}\ (\mathsf{h}.\ e_2)]\rangle_{\mathsf{eval}} \\
\langle \mathsf{throw}\ \mathit{err}\ e,\ K\rangle_{\mathsf{eval}} &\mapsto \langle e,\ K[\mathsf{throw}\ \mathit{err}\ \square]\rangle_{\mathsf{eval}} \\
\langle K[\mathsf{try}\ \square\ \mathsf{catch}\ \mathit{err}\ \mathsf{with}\ (\mathsf{h}.\ e)],\ v\rangle_{\mathsf{cont}} &\mapsto \langle v,\ K\rangle_{\mathsf{eval}} \\
\langle K'[\mathsf{try}\ K[\mathsf{throw}\ \mathit{err}\ \square]\ \mathsf{catch}\ \mathit{err}\ \mathsf{with}\ (\mathsf{h}.\ e)],\ v\rangle_{\mathsf{cont}} &\mapsto \langle e[v/\mathsf{h}],\ K'\rangle_{\mathsf{eval}} \\
&\quad (\mathsf{try}\ldots\mathsf{catch}\ \mathit{err}\ \ldots\mathsf{with} \notin K)
\end{aligned}$$

Fig. 13. Operational semantics of $\lambda_{\mathsf{rec,exc}}$.

follows:

$$\begin{aligned}
\langle C\rangle_{\mathsf{term}} &\mapsto \langle C,\ \square\rangle_{\mathsf{eval}} \\
&\mapsto \langle \mathsf{throw}\ \mathit{err}_1\ (\mathsf{throw}\ \mathit{err}_2\ 5),\ \mathsf{try}\ \square\ \mathsf{catch}\ \mathit{err}_2\ \mathsf{with}\ (\mathsf{h}.\ \mathsf{h} \oplus 1)\rangle_{\mathsf{eval}} \\
&\mapsto \langle \mathsf{throw}\ \mathit{err}_2\ 5,\ \mathsf{try}\ (\mathsf{throw}\ \mathit{err}_1\ \square)\ \mathsf{catch}\ \mathit{err}_2\ \mathsf{with}\ (\mathsf{h}.\ \mathsf{h} \oplus 1)\rangle_{\mathsf{eval}} \\
&\mapsto \langle 5,\ \mathsf{try}\ (\mathsf{throw}\ \mathit{err}_1\ (\mathsf{throw}\ \mathit{err}_2\ \square))\ \mathsf{catch}\ \mathit{err}_2\ \mathsf{with}\ (\mathsf{h}.\ \mathsf{h} \oplus 1)\rangle_{\mathsf{eval}} \\
&\mapsto \langle \mathsf{try}\ (\mathsf{throw}\ \mathit{err}_1\ (\mathsf{throw}\ \mathit{err}_2\ \square))\ \mathsf{catch}\ \mathit{err}_2\ \mathsf{with}\ (\mathsf{h}.\ \mathsf{h} \oplus 1),\ 5\rangle_{\mathsf{cont}} \\
&\mapsto \langle 5 \oplus 1,\ \square\rangle_{\mathsf{eval}} \mapsto^* \langle 6\rangle_{\mathsf{ret}}.
\end{aligned}$$

After entering the `catch`, each `throw` pushes a frame to evaluate its argument, so the nested payloads are forced from the inside out until the literal 5 is reached. Returning 5 to the innermost frame $\mathsf{throw}\ \mathit{err}_2\ \square$ raises err_2, so control unwinds to the nearest enclosing handler for err_2, discarding the pending $\mathsf{throw}\ \mathit{err}_1\ \square$ frame, and the handler increments the payload 5 to 6. (The outer $\mathsf{throw}\ \mathit{err}_1$ never fires: it could do so only once its argument $\mathsf{throw}\ \mathit{err}_2\ 5$ returned a value, which never happens.)

4.2 Modeling Exceptions in Guarded Interaction Trees

We now show how to model $\lambda_{\mathsf{rec,exc}}$ in GITrees, providing step-by-step guidance on the design decisions involved. This demonstrates the general methodology for modeling context-dependent effects.

Step 1: Identify the Key Operations. Looking at the operational semantics, we identify three key operations:

(1) *Registering a handler*: When entering try e_1 catch err with (h. e_2), we must remember that err can be caught by handler h;
(2) *Normal exit*: When e_1 completes without throwing, we must remove the handler;
(3) *Throwing an exception*: When throw err e is executed, we must find the nearest handler for err and transfer control.

Each of these will become an effect in our model.

Step 2: Design the State. What state is needed to implement these operations? We need to track the currently active exception handlers. Since handlers can nest (try blocks can contain other try blocks), we use a *stack* of handlers.

Each handler entry is a triple (err, h, κ) where:

- $err \in Exc$ is the exception name this handler catches;
- $h : \blacktriangleright\mathbf{IT} \to \blacktriangleright\mathbf{IT}$ is the handler function (taking the exception payload);
- $\kappa : \blacktriangleright\mathbf{IT} \to \blacktriangleright\mathbf{IT}$ is the *saved continuation* representing the computation outside the try block.

The stack is ordered with the most recent (innermost) handler at the head. When we register a handler, we need to remember *where to return* after the handler completes. The continuation κ represents all the computation that was "paused" when we entered the try block. Without it, after handling an exception, we would not know how to resume the surrounding computation.

This is analogous to how the operational semantics uses continuation stacks:
the evaluation context $K[\text{try } \square \text{ catch } err \text{ with } (\text{h. } e)]$ remembers both the handler h and the outer continuation K.

Step 3: Define the Effect Signatures. Based on our operations, we define three effects, shown in Figure 14:

- `register`: Takes an exception name err, handler h, and the computation x to run inside the try block. Returns the result of x. The reifier pushes (err, h, κ) onto the handler stack, where κ is the current continuation.
- `pop`: Takes an exception name err. Returns unit. The reifier removes the topmost handler for err from the stack, restoring the saved continuation.
- `throw`: Takes an exception name err and payload x. Never returns (output type is 0, the empty type). The reifier searches the stack for the nearest handler for err, applies the handler to the payload, and continues with the saved continuation.

Step 4: Implement the Reifiers. The reifier functions, shown in Figure 14, directly implement the stack operations:

- $r_{\texttt{register}}$ pushes a new handler onto the stack and returns the inner computation;
- $r_{\texttt{pop}}$ removes the handler from the stack if it matches the given exception name (failing otherwise), and continues with both the current and saved continuations;
- $r_{\texttt{throw}}$ searches for the first matching handler, applies it to the payload, and continues with the saved continuation, *discarding* the current continuation.

The fact that $r_{\texttt{throw}}$ searches the stack and discards intermediate continuations is what gives exceptions their non-local control transfer behavior.

Step 5: Define Convenient Wrapper Functions. We define high-level wrapper functions that combine the primitive effects:

- Throw(err, x): First evaluates x to a value, then performs `throw`;

$$\boxed{State \triangleq \mathsf{list}\ (Exc \times (\blacktriangleright\mathbf{IT} \to \blacktriangleright\mathbf{IT}) \times (\blacktriangleright\mathbf{IT} \to \blacktriangleright\mathbf{IT}))}$$

$$\begin{aligned} Ins_{\mathsf{throw}}(X) &\triangleq Exc \times \blacktriangleright X & Outs_{\mathsf{throw}}(X) &\triangleq 0 \\ Ins_{\mathsf{register}}(X) &\triangleq Exc \times (\blacktriangleright X \to \blacktriangleright X) \times \blacktriangleright X & Outs_{\mathsf{register}}(X) &\triangleq \blacktriangleright X \\ Ins_{\mathsf{pop}}(X) &\triangleq Exc & Outs_{\mathsf{pop}}(X) &\triangleq 1 \end{aligned}$$

$$r_{\mathsf{register}}((err, h, x), \sigma, \kappa) = \mathsf{Some}(x, (err, h, \kappa) :: \sigma)$$

$$r_{\mathsf{throw}}((err, x), \sigma, \kappa) = \begin{cases} \mathsf{Some}(\kappa'\ (h\ x), \sigma_2), & \sigma = \sigma_1 +\!\!+ [(err, h, \kappa')] +\!\!+ \sigma_2 \\ \mathsf{None}, & \text{otherwise} \end{cases}$$

$$r_{\mathsf{pop}}(err, \sigma, \kappa) = \begin{cases} \mathsf{None}, & \sigma = [] \\ \mathsf{None}, & \sigma = (err', h, \kappa') :: \sigma' \text{ and } err \neq err' \\ \mathsf{Some}(\kappa'\ (\kappa\ ()), \sigma'), & \sigma = (err, h, \kappa') :: \sigma' \end{cases}$$

$$\begin{aligned} \mathsf{Pop}(err) &\triangleq \mathsf{Vis}_{\mathsf{pop}}(err, \lambda_.\mathsf{next}(\mathsf{Ret}())) \\ \mathsf{Reg}(err, h, x) &\triangleq \mathsf{Vis}_{\mathsf{register}}((err, h, x), \mathsf{id}) \\ \mathsf{Throw}(err, x) &\triangleq \mathsf{get_val}(x, \lambda r.\mathsf{Vis}_{\mathsf{throw}}((err, \mathsf{next}(r)), \bot\text{-elim})) \\ \mathsf{Catch}(err, h, x) &\triangleq \mathsf{Reg}(err, h, \mathsf{next}(\mathsf{get_val}(x, \lambda r.\mathsf{get_val}(\mathsf{Pop}(err), \lambda_.r)))) \end{aligned}$$

Fig. 14. Effects for $\lambda_{\text{rec,exc}}$.

$$\begin{aligned} &\mathbf{E}[\![x]\!]_\rho = \rho(x) \\ &\mathbf{E}[\![n]\!]_\rho = \mathsf{Ret}(n) \\ &\mathbf{E}[\![\mathsf{rec}\ f(x) = e]\!]_\rho = \mathsf{Fun}(\mathsf{next}(\lambda a.\ \mathbf{E}[\![e]\!]_{\rho[f \mapsto \mathbf{E}[\![\mathsf{rec}\ f(x)=e]\!]_\rho,\ x \mapsto a]})) \\ &\mathbf{E}[\![e_1\ e_2]\!]_\rho = \mathbf{E}[\![e_1]\!]_\rho \bullet \mathbf{E}[\![e_2]\!]_\rho \\ &\mathbf{E}[\![e_1 \oplus e_2]\!]_\rho = \mathsf{NatOp}_\oplus(\mathbf{E}[\![e_1]\!]_\rho, \mathbf{E}[\![e_2]\!]_\rho) \\ &\mathbf{E}[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!]_\rho = \mathsf{If}(\mathbf{E}[\![e_1]\!]_\rho, \mathbf{E}[\![e_2]\!]_\rho, \mathbf{E}[\![e_3]\!]_\rho) \\ &\mathbf{E}[\![\mathsf{throw}\ err\ e]\!]_\rho = \mathsf{Throw}(err, \mathbf{E}[\![e]\!]_\rho) \\ &\mathbf{E}[\![\mathsf{try}\ e_1\ \mathsf{catch}\ err\ \mathsf{with}\ (\mathsf{h}.\ e_2)]\!]_\rho = \mathsf{Catch}(err, \lambda v.\mathbf{E}[\![e_2]\!]_{\rho[h \mapsto v]}, \mathbf{E}[\![e_1]\!]_\rho) \end{aligned}$$

Fig. 15. Denotational semantics for $\lambda_{\text{rec,exc}}$.

- $\mathsf{Catch}(err, h, x)$: Performs `register`, evaluates x, then performs pop to clean up the handler.

The Catch function shows a key pattern: we register the handler, run the protected computation, and then pop the handler on normal exit. If an exception is thrown during the computation, the `throw` effect will handle it, and the pop will never be reached.

Step 6: Define the Denotational Semantics. The interpretation of exception constructs, shown in Figure 15, is now straightforward.

Note how the handler $h.\ e_2$ becomes a lambda function that interprets e_2 with h bound to the exception payload.

$$\mathbf{K}[\![\Box]\!]_\rho = (\lambda x.\, x, [])$$
$$\mathbf{K}[\![K\ v]\!]_\rho = \text{let } (\kappa, \sigma) = \mathbf{K}[\![K]\!]_\rho \text{ in } \big(\kappa \circ (\lambda x.\, x \bullet (\mathbf{E}[\![v]\!]_\rho)), \sigma\big)$$
$$\mathbf{K}[\![e\ K]\!]_\rho = \text{let } (\kappa, \sigma) = \mathbf{K}[\![K]\!]_\rho \text{ in } \big(\kappa \circ (\lambda x.\, (\mathbf{E}[\![e]\!]_\rho) \bullet x), \sigma\big)$$
$$\mathbf{K}[\![e \oplus K]\!]_\rho = \text{let } (\kappa, \sigma) = \mathbf{K}[\![K]\!]_\rho \text{ in } \big(\kappa \circ (\lambda x.\, \mathrm{NatOp}_\oplus(\mathbf{E}[\![e]\!]_\rho, x)), \sigma\big)$$
$$\mathbf{K}[\![K \oplus v]\!]_\rho = \text{let } (\kappa, \sigma) = \mathbf{K}[\![K]\!]_\rho \text{ in } \big(\kappa \circ (\lambda x.\, \mathrm{NatOp}_\oplus(x, \mathbf{E}[\![v]\!]_\rho)), \sigma\big)$$
$$\mathbf{K}[\![\text{if } K \text{ then } e_1 \text{ else } e_2]\!]_\rho = \text{let } (\kappa, \sigma) = \mathbf{K}[\![K]\!]_\rho \text{ in } \big(\kappa \circ (\lambda x.\, \mathrm{If}(x, \mathbf{E}[\![e_1]\!]_\rho, \mathbf{E}[\![e_2]\!]_\rho)), \sigma\big)$$
$$\mathbf{K}[\![\text{throw } err\ K]\!]_\rho = \text{let } (\kappa, \sigma) = \mathbf{K}[\![K]\!]_\rho \text{ in } (\kappa \circ (\lambda x.\, \mathrm{Throw}(err, x)), \sigma)$$
$$\mathbf{K}[\![\text{try } K \text{ catch } err \text{ with } (\mathrm{h}.\ e)]\!]_\rho = \text{let } (\kappa, \sigma) = \mathbf{K}[\![K]\!]_\rho \text{ in}$$
$$\big(\lambda x.\, \mathrm{get_val}(x, \lambda y.\, \mathrm{get_val}(\mathrm{Pop}(err), \lambda_.\, y)), (err, \blacktriangleright(\lambda x.\, \mathbf{E}[\![e]\!]_{h \mapsto x}), \blacktriangleright\kappa) :: \sigma\big)$$

$$\mathbf{S}[\![\langle e,\ K\rangle_{\mathsf{eval}}]\!]_\rho = \text{let } (\kappa, \sigma) = \mathbf{K}[\![K]\!]_\rho \text{ in } \big(\kappa(\mathbf{E}[\![e]\!]_\rho), \sigma\big)$$
$$\mathbf{S}[\![\langle K,\ v\rangle_{\mathsf{cont}}]\!]_\rho = \text{let } (\kappa, \sigma) = \mathbf{K}[\![K]\!]_\rho \text{ in } \big(\kappa(\mathbf{E}[\![v]\!]_\rho), \sigma\big)$$
$$\mathbf{S}[\![\langle e\rangle_{\mathsf{term}}]\!]_\rho = \big(\mathbf{E}[\![e]\!]_\rho, []\big)$$
$$\mathbf{S}[\![\langle v\rangle_{\mathsf{ret}}]\!]_\rho = \big(\mathbf{E}[\![v]\!]_\rho, []\big)$$

Fig. 16. Denotational semantics of the abstract machine's configurations.

We then proceed to lift $\mathbf{E}[\![-]\!]$ to evaluation contexts, $\mathbf{K}[\![-]\!]$, and the abstract machine's configurations, $\mathbf{S}[\![-]\!]$, in Figure 16.

Step 7: Prove Soundness. We prove that the denotational semantics is sound with respect to the operational semantics:

Theorem 4.1 (Soundness). *Let $c_0, c_1 \in \mathit{Config}$ and suppose $c_0 \mapsto c_1$. Then $\mathbf{S}[\![c_0]\!]_\rho \rightsquigarrow^* \mathbf{S}[\![c_1]\!]_\rho$ for all ρ.*

The proof proceeds by induction on the operational semantics derivation, using the program logic rules derived in the next paragraph.

Step 8: Derive Program Logic Rules. From the effect signatures and reifiers, we derive program logic rules for reasoning about exceptions, shown in Figure 17. These rules follow directly from the wp-reify rule for general effects. The key rules are:

- wp-reg: Registering a handler requires the state σ and produces state with the handler pushed: $(err, h, \blacktriangleright\kappa) :: \sigma$;
- wp-throw': Throwing requires that a matching handler exists in the state, and transfers control to that handler;
- wp-catch-throw: A combined rule for the common case where an exception is immediately thrown and caught within the same Catch. Here the inner context κ' is required to be a homomorphism: since installing a handler places the protected computation in the input of a `register` effect, a homomorphic κ' cannot register a nearer handler for err between the Catch and the throw, so the throw is indeed caught by this Catch's handler h.

These rules enable modular reasoning about exception-using code within the program logic.

WP-REG

$$\frac{\kappa \in \mathit{Hom} \quad \mathsf{has_state}(\sigma) \quad \triangleright(\mathsf{has_state}((\mathit{err}, h, \blacktriangleright\kappa) :: \sigma) \mathrel{-\!\!*} \mathsf{wp}\ \beta\ \{\Phi\})}{\mathsf{wp}\ (\kappa(\mathsf{Reg}(\mathit{err}, h\ (\mathsf{next}(\beta)))))\ \{\Phi\}}$$

WP-POP'

$$\frac{\kappa, \kappa' \in \mathit{Hom} \quad \mathsf{has_state}((\mathit{err}, h, \kappa') :: \sigma) \quad \triangleright(\mathsf{has_state}(\sigma) \mathrel{-\!\!*} \mathsf{wp}\ \beta\ \{\Phi\}) \quad \kappa'(\blacktriangleright\kappa\ (\mathsf{next}(\mathsf{Ret}()))) = \mathsf{next}(\beta)}{\mathsf{wp}\ (\kappa(\mathsf{Pop}(\mathit{err})))\ \{\Phi\}}$$

WP-CATCH

$$\frac{\kappa \in \mathit{Hom} \quad \mathsf{has_state}(\sigma) \quad x \in \mathit{Val} \quad \triangleright(\mathsf{has_state}(\sigma) \mathrel{-\!\!*} \mathsf{wp}\ (\kappa\ x)\ \{\Phi\})}{\mathsf{wp}\ (\kappa(\mathsf{Catch}(\mathit{err}, h, x)))\ \{\Phi\}}$$

WP-THROW'

$$\frac{\kappa, \kappa' \in \mathit{Hom} \quad \mathit{err} \notin \sigma' \quad \mathsf{has_state}(\sigma' +\!\!+ [(\mathit{err}, h, \kappa')] +\!\!+ \sigma) \quad \triangleright(\mathsf{has_state}(\sigma) \mathrel{-\!\!*} \mathsf{wp}\ \beta\ \{\Phi\}) \quad \kappa'(h\ (\mathsf{next}(x))) = \mathsf{next}(\beta)}{\mathsf{wp}\ (\kappa(\mathsf{Throw}(\mathit{err}, x)))\ \{\Phi\}}$$

WP-CATCH-THROW

$$\frac{\kappa, \kappa' \in \mathit{Hom} \quad \mathsf{has_state}(\sigma) \quad \triangleright(\mathsf{has_state}(\sigma) \mathrel{-\!\!*} \mathsf{wp}\ (\kappa\ \beta)\ \{\Phi\}) \quad h\ (\mathsf{next}(x)) = \mathsf{next}(\beta)}{\mathsf{wp}\ (\kappa(\mathsf{Catch}(\mathit{err}, h, (\kappa'(\mathsf{Throw}(\mathit{err}, x))))))\ \{\Phi\}}$$

Fig. 17. Weakest precondition rules for exceptions.

Summary. This completes our treatment of exceptions. We have shown how to systematically design effects for exceptions by:

(1) Identifying the key operations from the operational semantics;
(2) Designing appropriate state (a handler stack with saved continuations);
(3) Defining effect signatures that capture these operations;
(4) Implementing reifiers that manipulate the state correctly;
(5) Deriving high-level wrappers and program logic rules.

This methodology generalizes to other context-dependent effects. The key is understanding what state needs to be maintained and how the effects should manipulate it. In the next section, we apply similar techniques to the more complex case of delimited continuations.

5 MODELING DELIMITED CONTINUATIONS

In this section we show how guarded interaction trees can be used to give semantics to delimited continuations which are a challenging example of context-dependent effects. We provide a denotational semantics for a programming language λ_{delim} with `shift` and `reset` effects. Our approach here for giving denotational semantics to λ_{delim} is similar to what we did for $\lambda_{\mathsf{rec,exc}}$ in Section 4. However, in addition to soundness of our denotational semantics (Theorem 4.1), we also show computational adequacy for λ_{delim} relative to an abstract machine semantics [7]. To the best of our knowledge, this represents the first formalized sound and adequate direct-style denotational semantics for delimited continuations.

$$\boxed{\Gamma;\alpha \vdash e : \tau;\beta \quad \Gamma \vdash_{\text{pure}} e : \tau}$$

$$\begin{array}{lll}
\text{types} & Ty \ni \tau,\sigma,\alpha,\beta,\delta,\gamma & ::= \mathbb{N} \mid \tau/\alpha \to \sigma/\beta \mid cont(\tau,\alpha) \\
\text{expressions} & Expr \ni e & ::= v \mid x \mid e_1\ e_2 \mid e_1 \oplus e_2 \\
 & & \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \mathcal{S}\ \mathsf{x}.\ e \mid \mathcal{D}\ \mathsf{e} \mid e_1\ @\ e_2 \\
\text{values} & Val \ni v & ::= n \mid \text{rec } f(x) = e \mid \text{cont } K \\
\text{eval. contexts} & Ectx \ni K & ::= \square \mid K[\text{if } \square \text{ then } e_1 \text{ else } e_2] \mid K[v\ \square] \mid K[\square\ e] \\
 & & \mid K[e \oplus \square] \mid K[\square \oplus v] \mid K[\square\ @\ v] \mid K[e\ @\ \square]
\end{array}$$

$$\frac{\Gamma \vdash_{\text{pure}} e : \tau}{\Gamma;\alpha \vdash e : \tau;\alpha} \qquad \frac{\Gamma, x : \text{cont}\ (\tau,\alpha);\sigma \vdash e : \sigma;\beta}{\Gamma;\alpha \vdash \mathcal{S}\ \mathsf{x}.\ e : \tau;\beta} \qquad \frac{\Gamma;\tau \vdash e : \tau;\sigma}{\Gamma \vdash_{\text{pure}} \mathcal{D}\ \mathsf{e} : \sigma} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash_{\text{pure}} x : \tau}$$

$$\frac{\Gamma, f : \sigma/\alpha \to \tau/\beta, x : \sigma;\alpha \vdash e : \tau;\beta}{\Gamma \vdash_{\text{pure}} \text{rec } f(x) = e : \sigma/\alpha \to \tau/\beta} \qquad \frac{\Gamma;\gamma \vdash e_1 : \sigma/\alpha \to \tau/\beta;\delta \quad \Gamma;\beta \vdash e_2 : \sigma;\gamma}{\Gamma;\alpha \vdash e_1\ e_2 : \tau;\delta}$$

$$\frac{\Gamma;\beta \vdash e_1 : \mathbb{N};\alpha \quad \Gamma;\sigma \vdash e_2 : \tau;\beta \quad \Gamma;\sigma \vdash e_3 : \tau;\beta}{\Gamma;\sigma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau;\alpha} \qquad \frac{}{\Gamma \vdash_{\text{pure}} n : \mathbb{N}}$$

$$\frac{\Gamma;\alpha \vdash e_1 : \mathbb{N};\beta \quad \Gamma;\beta \vdash e_2 : \mathbb{N};\sigma}{\Gamma;\alpha \vdash e_1 \oplus e_2 : \mathbb{N};\sigma} \qquad \frac{\Gamma;\sigma \vdash e_1 : \text{cont}\ (\tau,\alpha);\delta \quad \Gamma;\delta \vdash e_2 : \tau;\beta}{\Gamma;\sigma \vdash e_1\ @\ e_2 : \alpha;\beta}$$

Fig. 18. Syntax and typing rules of λ_{delim}.

5.1 Syntax and Operational Semantics of λ_{delim}

The syntax and the type system of λ_{delim} is given in Figure 18. It is similar to λ_{callcc}, but instead of call/cc $(-.\ -)$ there are operators $\mathcal{D}$ e (delimit the current evaluation context, also known as `reset`) and $\mathcal{S}$ x. e (grab the current delimited continuation, and bind it to x in e, also known as `shift`).

The type system follows Danvy and Filinski [12], extending simply-typed λ-calculus with answer types α, β. The main typing judgment $\Gamma;\alpha \vdash e : \tau;\beta$ means: under the typing context Γ, expression e can be plugged into a context expecting a value of type τ and producing a value of type α; in that case the resulting program will have the type β. You can think of it a computation $\Gamma \to (\tau \to \alpha) \to \beta$ under the CPS translation. Thus, the type of the (delimited) continuation corresponds to $\tau \to \alpha$, while the type of the overall expression is β. The pure typing judgment $\Gamma \vdash_{\text{pure}} e : \tau$ indicates e does not depend on the surrounded context, and is context-independent for any answer types. Expressions can change their context's answer type, as seen in the $\mathcal{S}$ x. e typing rule.

For example, suppose we extend the type system with booleans, $\mathbb{B}$, and add a primitive function isprime that does not modify answer types. That is $\emptyset;\alpha \vdash \text{isprime} : \mathbb{N}/\beta \to \mathbb{B}/\beta;\alpha$. The expression $\mathcal{D}$ ((rec f(x) = isprime ($\mathcal{S}$ k. x − 1)) 2) is well-typed in this type system as a $\mathbb{N}$, even though it changes the answer type from $\mathbb{B}$ to $\mathbb{N}$.

Answer types appear in both judgments and type constructors. Continuation type cont (τ,α) represents contexts expecting something of the type τ and producing something of the type α. Function type $\sigma/\alpha \to \tau/\beta$, in addition to the input type σ and the output type τ, record the typing of the surrounding context at the point of the function call. See [12] for details.

The operational semantics for λ_{delim}, similarly to $\lambda_{\text{rec,exc}}$, uses a CEK machine, following [6, 7, 17]. Selected reduction rules appear in Figure 19 (see Rocq formalization for the full set of rules). The

metacont. $\mathit{Mcont} \ni \mathit{mk} ::= [] \mid K :: \mathit{mk}$

config. $\mathit{Config} ::= \langle e,\ K,\ \mathit{mk}\rangle_{\mathsf{eval}} \mid \langle K,\ v,\ \mathit{mk}\rangle_{\mathsf{cont}} \mid \langle \mathit{mk},\ v\rangle_{\mathsf{mcont}} \mid \langle e\rangle_{\mathsf{term}} \mid \langle v\rangle_{\mathsf{ret}}$

$$
\begin{aligned}
\langle e\rangle_{\mathsf{term}} &\mapsto \langle e,\ \Box,\ []\rangle_{\mathsf{eval}} \\
\langle K :: \mathit{mk},\ v\rangle_{\mathsf{mcont}} &\mapsto \langle K,\ v,\ \mathit{mk}\rangle_{\mathsf{cont}} \\
\langle [],\ v\rangle_{\mathsf{mcont}} &\mapsto \langle v\rangle_{\mathsf{ret}} \\
\langle \Box,\ v,\ \mathit{mk}\rangle_{\mathsf{cont}} &\mapsto \langle \mathit{mk},\ v\rangle_{\mathsf{mcont}} \\
\langle v,\ K,\ \mathit{mk}\rangle_{\mathsf{eval}} &\mapsto \langle K,\ v,\ \mathit{mk}\rangle_{\mathsf{cont}} \\
\langle e_0\ e_1,\ K,\ \mathit{mk}\rangle_{\mathsf{eval}} &\mapsto \langle e_0,\ K[\Box\ e_1],\ \mathit{mk}\rangle_{\mathsf{eval}} \\
\langle e_0 \oplus e_1,\ K,\ \mathit{mk}\rangle_{\mathsf{eval}} &\mapsto \langle e_1,\ K[e_0 \oplus \Box],\ \mathit{mk}\rangle_{\mathsf{eval}} \\
\langle \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3,\ K,\ \mathit{mk}\rangle_{\mathsf{eval}} &\mapsto \langle e_1,\ K[\mathsf{if}\ \Box\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3],\ \mathit{mk}\rangle_{\mathsf{eval}} \\
\langle e_0\ @\ e_1,\ K,\ \mathit{mk}\rangle_{\mathsf{eval}} &\mapsto \langle e_1,\ K[e_0\ @\ \Box],\ \mathit{mk}\rangle_{\mathsf{eval}} \\
\langle \mathcal{D}\ \mathsf{e},\ K,\ \mathit{mk}\rangle_{\mathsf{eval}} &\mapsto \langle e,\ \Box,\ K :: \mathit{mk}\rangle_{\mathsf{eval}} \\
\langle \mathcal{S}\ \mathsf{k}.\ e,\ K,\ \mathit{mk}\rangle_{\mathsf{eval}} &\mapsto \langle e[\mathsf{cont}\ K/k],\ \Box,\ \mathit{mk}\rangle_{\mathsf{eval}} \\
\langle K[\Box\ e],\ v,\ \mathit{mk}\rangle_{\mathsf{cont}} &\mapsto \langle e,\ K[v\ \Box],\ \mathit{mk}\rangle_{\mathsf{eval}} \\
\langle K[(\mathsf{rec}\ f(x) = e)\ \Box],\ v,\ \mathit{mk}\rangle_{\mathsf{cont}} &\mapsto \langle e[v/x][\mathsf{rec}\ f(x) = e/f],\ K,\ \mathit{mk}\rangle_{\mathsf{eval}} \\
\langle K[e \oplus \Box],\ v,\ \mathit{mk}\rangle_{\mathsf{cont}} &\mapsto \langle e,\ K[\Box \oplus v],\ \mathit{mk}\rangle_{\mathsf{eval}} \\
\langle K[\Box \oplus v_1],\ v_0,\ \mathit{mk}\rangle_{\mathsf{cont}} &\mapsto \langle v_2,\ K,\ \mathit{mk}\rangle_{\mathsf{eval}} \quad (v_0 \oplus v_1 = v_2) \\
\langle K[\mathsf{if}\ \Box\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3],\ n,\ \mathit{mk}\rangle_{\mathsf{cont}} &\mapsto \langle e_3,\ K,\ \mathit{mk}\rangle_{\mathsf{eval}} \quad (n = 0) \\
\langle K[\mathsf{if}\ \Box\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3],\ n,\ \mathit{mk}\rangle_{\mathsf{cont}} &\mapsto \langle e_2,\ K,\ \mathit{mk}\rangle_{\mathsf{eval}} \quad (0 < n) \\
\langle K[e\ @\ \Box],\ v,\ \mathit{mk}\rangle_{\mathsf{cont}} &\mapsto \langle e,\ K[\Box\ @\ v],\ \mathit{mk}\rangle_{\mathsf{eval}} \\
\langle K[\Box\ @\ v],\ \mathsf{cont}\ K',\ \mathit{mk}\rangle_{\mathsf{cont}} &\mapsto \langle K',\ v,\ K :: \mathit{mk}\rangle_{\mathsf{cont}}
\end{aligned}
$$

Fig. 19. Operational semantics of λ_{delim}.

abstract machine operates on various *configurations*, which can be of several forms. The first one is the initial configuration $\langle e\rangle_{\mathsf{term}}$, which is just a starting state for evaluating expressions. Similarly, there is a terminal configuration $\langle v\rangle_{\mathsf{ret}}$ signifying that the program has terminated with the value v.

From the initial configuration, we go on to $\langle e,\ K,\ \mathit{mk}\rangle_{\mathsf{eval}}$, which signifies that we are evaluating an expression e inside the current delimited context K, with the metacontinuation mk (a stack of continuations based on different delimiters). It is this configuration type which takes care of delimited control operations. The $\mathcal{D}$ operator saves the current continuation on top of the metacontinuation, limiting the scope of $\mathcal{S}$ x. e. The $\mathcal{S}$ x. e operation behaves similarly to call/cc (x. e), except that it prevents later control operators from capturing its evaluation context.

The last two configuration types are for dealing with continuations and metacontinuations. A configuration $\langle K,\ v,\ \mathit{mk}\rangle_{\mathsf{cont}}$ signifies that we are trying to plug in the value v into the context K, with the metacontinuation mk. A configuration $\langle \mathit{mk},\ v\rangle_{\mathsf{mcont}}$ signifies that we are done with the current continuation (ending with the value v), but we still have to unwind the continuation stack mk.

$$
\begin{aligned}
Ins_{\mathtt{reset}}(X) &\triangleq \blacktriangleright X & Ins_{\mathtt{shift}}(X) &\triangleq (\blacktriangleright X \to \blacktriangleright X) \to \blacktriangleright X \\
Ins_{\mathsf{pop}}(X) &\triangleq \blacktriangleright X & Ins_{\mathtt{appcont}}(X) &\triangleq \blacktriangleright X \times \blacktriangleright (X \to X) \\
Outs_{\mathtt{reset}}(X) &\triangleq \blacktriangleright X & Outs_{\mathtt{shift}}(X) &\triangleq \blacktriangleright X \\
Outs_{\mathsf{pop}}(X) &\triangleq 0 & Outs_{\mathtt{appcont}}(X) &\triangleq \blacktriangleright X \\
r_{\mathtt{reset}}(e, \sigma, \kappa) &= \mathsf{Some}(e, \kappa :: \sigma) & r_{\mathtt{shift}}(f, \sigma, \kappa) &= \mathsf{Some}(f\ \kappa, \sigma) \\
r_{\mathsf{pop}}(e, [], _) &= \mathsf{Some}(e, []) & r_{\mathsf{pop}}(e, \kappa :: \sigma, _) &= \mathsf{Some}(\kappa\ e, \sigma) \\
r_{\mathtt{appcont}}((e, \kappa), \sigma, \kappa') &= \mathsf{Some}(\kappa\ e, \kappa' :: \sigma) & \mathbf{P}(\beta) &\triangleq \mathsf{get_val}(\beta, \mathsf{Pop}) \\
\mathsf{Reset}(e) &\triangleq \mathsf{Vis}_{\mathtt{reset}}(e, \mathsf{id}) & \mathsf{Shift}(f) &\triangleq \mathsf{Vis}_{\mathtt{shift}}(f, \mathsf{id}) \\
\mathsf{Appcont}(e, f) &\triangleq \mathsf{Vis}_{\mathtt{appcont}}((e, f), \mathsf{id}) & \mathsf{Pop}(e) &\triangleq \mathsf{Vis}_{\mathsf{pop}}(e, \lambda x.\, \mathsf{abort}\ x)
\end{aligned}
$$

Fig. 20. Effects for λ_{delim}.

5.2 Denotational Semantics of λ_{delim}

Our model represents delimited continuations with effects mimicking an abstract machine, operating on semantic rather than syntactic components. The effect signature and reifiers (Figure 20) define a state with a stack of continuations, manipulated explicitly. The effect signature $E_{\lambda_{\mathsf{delim}}}$ includes four operators: {`reset`, `shift`, pop, `appcont`}. The signature of `reset` simply tells us that the corresponding effect does not directly modify its argument. The auxiliary effect pop, which does not have an equivalent in the surface syntax, is used to enforce unwinding of the continuation stack. As the output arity of pop signifies, it does not return. We describe the importance of that below. The rest of the signature is more straightforward: `shift` and `appcont` are defined exactly as `callcc` and `throw`. The semantics of these effects, in terms of reification, is more intricate. As we mentioned, the state for reification is $\mathit{State} = \mathit{List}(\blacktriangleright\mathbf{IT} \to \blacktriangleright\mathbf{IT})$.

In comparison with call/cc (x. e), the control operator $\mathcal{S}$ x. e does not necessarily continue from the same continuation; hence, the corresponding reifier passes the current continuation to the body, but does not return control back. The reifier for `reset` simply saves the current continuation κ onto the stack σ. It is then the job of the pop operation to restore the continuation from the stack. The reifier for `shift` is similar to that of `callcc`, except that it removes the current continuation entirely. The reifier for `appcont`, in comparison with `throw`, does not simply pass control, but also saves the current continuation on the stack. This corresponds to the fact that whenever a delimited continuation is invoked, the result is wrapped in a reset; that is done to prevent the continuation from escaping the delimiter. As part of instantiating GITrees with these effects, we obtain the specialized program logic rules shown in Figure 21. We will use those rules later for defining a logical relation between the syntax and the semantics of λ_{delim}. As mentioned above, we will use Pop to unwind the continuation stack and restore the continuation after finishing with a `reset`. This means that we will need to insert explicit calls to Pop in the interpretation of λ_{delim}. For these purposes, we use an abbreviation $\mathbf{P}(\beta)$, which first evaluates β to a value, and then executes the pop operation.

The interpretation of λ_{delim} uses this auxiliary function and is given in Figure 22. Similarly to the operational semantics, the interpretation is divided into five categories. First, we have $\mathbf{E}[\![-]\!]$ and $\mathbf{V}[\![-]\!]$ for the interpretation of expressions and values, which is what we need for the surface syntax. All of those interpretations return GITrees. Note that in the interpretation of $\mathcal{D}$ − we insert explicit calls to $\mathbf{P}$, and similarly in the interpretation of continuations.

$$\textsc{wp-shift}\quad \frac{\mathsf{has_state}(\sigma) \quad \triangleright(\mathsf{has_state}(\sigma) \mathrel{-\!\!*} \mathsf{wp}\ \beta\ \{\Phi\}) \quad \blacktriangleright \mathbf{P}(f(\blacktriangleright\kappa)) = \mathsf{next}(\beta)}{\mathsf{wp}\ (\kappa(\mathsf{Shift}(f)))\ \{\Phi\}}$$

$$\textsc{wp-reset}\quad \frac{\mathsf{has_state}(\sigma) \quad \triangleright(\mathsf{has_state}(\blacktriangleright\kappa :: \sigma) \mathrel{-\!\!*} \mathsf{wp}\ \mathbf{P}(e)\ \{\Phi\})}{\mathsf{wp}\ (\kappa(\mathsf{Reset}(\mathsf{next}(e))))\ \{\Phi\}}$$

$$\textsc{wp-pop}\quad \frac{\mathsf{has_state}(\sigma) \quad \kappa' = \kappa \text{ if } \sigma = \kappa :: \sigma' \text{ and id otherwise} \quad \triangleright(\mathsf{has_state}(\mathtt{tail}(\sigma)) \mathrel{-\!\!*} \mathsf{wp}\ (\kappa'(v))\ \{\Phi\})}{\mathsf{wp}\ \mathbf{P}(v)\ \{\Phi\}}$$

$$\textsc{wp-appcont}\quad \frac{\mathsf{has_state}(\sigma) \quad \triangleright(\mathsf{has_state}(\blacktriangleright\kappa :: \sigma) \mathrel{-\!\!*} \mathsf{wp}\ \beta\ \{\Phi\}) \quad \blacktriangleright\kappa'(e) = \mathsf{next}(\beta)}{\mathsf{wp}\ (\kappa(\mathsf{Appcont}(e, \kappa')))\ \{\Phi\}}$$

Fig. 21. Weakest precondition rules for delimited continuations.

The other group of interpretations, $\mathbf{K}[\![-]\!]$, $\mathbf{M}[\![-]\!]$ and $\mathbf{S}[\![-]\!]$, are for interpreting continuations, metacontinuations, and other configurations; these are used for showing soundness (preservation of operational semantics by the interpretation). The interpretation $\mathbf{K}[\![-]\!]$ of continuations returns a semantic continuation (a function $\mathbf{IT} \to \mathbf{IT}$). Similarly, the interpretations $\mathbf{M}[\![-]\!]$ (resp. $\mathbf{S}[\![-]\!]$) of metacontinuations (resp. configurations) returns a stack of semantic continuations (resp. a semantic configuration).

We now show that our interpretation is sound w.r.t. the abstract machine semantics. For this we prove lemmas similar to Lemma 3.1, and put them to use in the soundness theorem:

Theorem 5.1. *Soundness. Let $c_0, c_1 \in \mathit{Config}$ and suppose $c_0 \mapsto c_1$. Then $\mathbf{S}[\![c_0]\!] \rightsquigarrow^* \mathbf{S}[\![c_1]\!]$.*

5.3 Logical Relation and Adequacy

We now show that our denotational semantics is adequate with regards to the abstract machine semantics. Specifically, we show the following result:

Theorem 5.2. *Adequacy. Suppose $\emptyset; \mathbb{N} \vdash e : \mathbb{N}; \mathbb{N}$ is a well-typed term, and that $(\mathbf{P}(\mathbf{E}[\![e]\!]_\emptyset), [\,]) \rightsquigarrow^* (\mathrm{Ret}(n), \sigma)$ for a natural number n and a metacontinuation σ. Then $\langle e \rangle_{\mathsf{term}} \mapsto^* \langle n \rangle_{\mathsf{ret}}$.*

We prove adequacy using a logical relation. It relates expressions to their interpretations and also connects syntactic and semantic configurations. The logical relation is shown in Figure 23. It is again a form of biorthogonal logical relation, with the main focus being the observational refinement O: two configurations are related if they reduce to the same natural number. This coincides with what we want to show in Theorem 5.2. To facilitate this we then lift O to the levels of metacontinuations, continuations and expressions. The relation $\mathcal{M}(P)$, where P : VRel is a relation on values, states that two metacontinuations are related if, whenever we plug in P-related values, the resulting configurations become O-related. Both $\mathcal{M}$ and O are then used to define the relation between semantic and syntactic continuations. The relation $\mathcal{K}(Q, P)$, where P, Q : VRel are relations on values, states that two continuations are related if, whenever we plug them into Q-related metacontinuations with P-related values, the resulting configurations become O-related. Finally, we use $\mathcal{K}$, $\mathcal{M}$, and O to define the relation between GITrees and λ_{delim} terms. The relation $\mathcal{E}(P, Q, R)$, where P, Q, R : VRel are relations on values, states that β is related to e if, whenever we plug them into (P, Q)-related continuations and an R-related metacontinuation, the resulting configurations become O-related.

The relations $\mathcal{E}$ and $\mathcal{K}$ are used to give semantics $[\![\tau]\!]$ to types. The idea is that $\mathcal{E}([\![\tau]\!], [\![\alpha]\!], [\![\beta]\!])$ relates terms $\emptyset; \alpha \vdash e : \tau; \beta$ to their semantic counterparts. This is then used, as expected for

$$
\begin{aligned}
&\mathbf{E}[\![x]\!]_\rho = \rho(x)\\
&\mathbf{E}[\![v]\!]_\rho = \mathbf{V}[\![v]\!]_\rho\\
&\mathbf{E}[\![e_1\ e_2]\!]_\rho = \mathsf{get_val}(\mathbf{E}[\![e_1]\!]_\rho, \lambda f.\ \mathsf{get_val}(\mathbf{E}[\![e_2]\!]_\rho, \lambda x.\ f \bullet x))\\
&\mathbf{E}[\![e_1 \oplus e_2]\!]_\rho = \mathsf{NatOp}_\oplus(\mathbf{E}[\![e_1]\!]_\rho, \mathbf{E}[\![e_2]\!]_\rho)\\
&\mathbf{E}[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!]_\rho = \mathsf{If}(\mathbf{E}[\![e_1]\!]_\rho, \mathbf{E}[\![e_2]\!]_\rho, \mathbf{E}[\![e_3]\!]_\rho)\\
&\mathbf{E}[\![\mathcal{D}\ \mathrm{e}]\!]_\rho = \mathsf{Reset}(\mathbf{P}(\mathbf{E}[\![e]\!]_\rho))\\
&\mathbf{E}[\![\mathcal{S}\ \mathrm{x}.\ e]\!]_\rho = \mathsf{Shift}(\mathbf{P} \circ (\lambda\kappa.\, \mathbf{E}[\![e]\!]_{\rho, x \mapsto \mathsf{Fun}(\mathsf{next}(\lambda y.\, \mathsf{Tau}(\kappa(\mathsf{next}\, y))))}))\\
&\mathbf{E}[\![e_1\ @\ e_2]\!]_\rho = \mathsf{get_val}(\mathbf{E}[\![e_2]\!]_\rho, \lambda x.\, \mathsf{get_fun}(\mathbf{E}[\![e_1]\!]_\rho, \lambda y.\, \mathsf{Appcont}(\mathsf{next}(x), y)))\\
&\mathbf{V}[\![n]\!]_\rho = \mathsf{Ret}(n)\\
&\mathbf{V}[\![\mathsf{rec}\, f(x) = e]\!]_\rho = \mathsf{Fun}(\mathsf{next}(\lambda a.\ \mathbf{E}[\![e]\!]_{\rho[f \mapsto \mathbf{V}[\![\mathsf{rec}\, f(x)=e]\!]_\rho,\ x \mapsto a]}))\\
&\mathbf{V}[\![\mathsf{cont}\ K]\!]_\rho = \mathsf{Fun}(\mathsf{next}(\lambda x.\, \mathsf{Tick}(\mathbf{P}(\mathbf{K}[\![K]\!]_\rho\ x))))\\
&\mathbf{K}[\![\square]\!]_\rho = \lambda x.\ x\\
&\mathbf{K}[\![K[\square\ e]]\!]_\rho = \lambda x.\ \mathbf{K}[\![K]\!]_\rho(\mathbf{E}[\![x\ e]\!]_\rho)\\
&\mathbf{K}[\![K[v\ \square]]\!]_\rho = \lambda x.\ \mathbf{K}[\![K]\!]_\rho(\mathbf{E}[\![v\ x]\!]_\rho)\\
&\mathbf{K}[\![K[e \oplus \square]]\!]_\rho = \lambda x.\ \mathbf{K}[\![K]\!]_\rho(\mathbf{E}[\![e \oplus x]\!]_\rho)\\
&\mathbf{K}[\![K[\square \oplus v]]\!]_\rho = \lambda x.\ \mathbf{K}[\![K]\!]_\rho(\mathbf{E}[\![x \oplus v]\!]_\rho)\\
&\mathbf{K}[\![K[\mathsf{if}\ \square\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]]\!]_\rho = \lambda x.\ \mathbf{K}[\![K]\!]_\rho(\mathbf{E}[\![\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]_\rho)\\
&\mathbf{K}[\![K[e\ @\ \square]]\!]_\rho = \lambda x.\ \mathbf{K}[\![K]\!]_\rho(\mathbf{E}[\![e\ @\ x]\!]_\rho)\\
&\mathbf{K}[\![K[\square\ @\ v]]\!]_\rho = \lambda x.\, \mathbf{K}[\![K]\!]_\rho(\mathbf{E}[\![x\ @\ v]\!]_\rho)\\
&\mathbf{M}[\![mk]\!]_\rho = \mathrm{map}(\lambda k.\, \mathbf{P} \circ \mathbf{K}[\![k]\!]_\rho)mk\\
&\mathbf{S}[\![\langle e,\ K,\ mk\rangle_{\mathsf{eval}}]\!]_\rho = \big(\mathbf{P}(\mathbf{E}[\![K[e]]\!]_\rho), \mathbf{M}[\![mk]\!]_\rho\big)\\
&\mathbf{S}[\![\langle K,\ v,\ mk\rangle_{\mathsf{cont}}]\!]_\rho = \big(\mathbf{P}(\mathbf{E}[\![K[v]]\!]_\rho), \mathbf{M}[\![mk]\!]_\rho\big)\\
&\mathbf{S}[\![\langle mk,\ v\rangle_{\mathsf{mcont}}]\!]_\rho = \big(\mathbf{P}(\mathbf{V}[\![v]\!]_\rho), \mathbf{M}[\![mk]\!]_\rho\big)\\
&\mathbf{S}[\![\langle e\rangle_{\mathsf{term}}]\!]_\rho = \big(\mathbf{P}(\mathbf{E}[\![e]\!]_\rho), []\big)\\
&\mathbf{S}[\![\langle v\rangle_{\mathsf{ret}}]\!]_\rho = \big(\mathbf{V}[\![v]\!]_\rho, []\big)
\end{aligned}
$$

Fig. 22. Denotational semantics for a calculus with delimited control.

logical relations, for defining the logical relation for function types and for open terms. The relation $[\![\Gamma]\!](\rho, \gamma)$ relates the semantic environment $\rho : \mathit{Var} \to \mathbf{IT}$ to the syntactic substitution $\gamma : \mathit{Var} \to \mathit{Expr}$; they are related if they map the same variables to related GITrees/expressions. Then we say that an expression e is semantically valid, $\Gamma; \alpha \vdash e : \tau; \beta$, if its interpretation $\mathbf{E}[\![e]\!]_\rho$ is related to $e[\gamma]$ under related substitutions ρ, γ. Note that if we ignore the answer types we can see that the logical relation exhibits a lot of similarities to the logical relation we gave in Section 3.3, and follows the same roadmap.

For this logical relation we obtain the fundamental property, which we will use for the proof of adequacy.

Lemma 5.3. *Fundamental lemma. Let* $\Gamma; \alpha \vdash e : \tau; \beta$ *then* $\Gamma; \alpha \vDash e : \tau; \beta$*; and if* $\Gamma \vdash_{\mathsf{pure}} e : \tau$ *then* $\Gamma \vDash_{\mathsf{pure}} e : \tau$.

$$
\begin{aligned}
\mathrm{SynConf} &\triangleq \mathit{Expr} \times \mathit{Ectx} \times \mathit{Mcont} & \mathrm{VRel} &\triangleq \mathbf{IT}^{v} \times \mathit{Val} \to \mathit{iProp}\\
\mathrm{SemConf} &\triangleq \mathbf{IT} \times \mathit{Hom} \times \mathit{List}(\mathit{Hom}) & \mathcal{O} &: \mathrm{ConfRel}\\
\mathrm{ConfRel} &\triangleq \mathrm{SemConf} \times \mathrm{SynConf} \to \mathit{iProp} & \mathcal{M} &: \mathrm{VRel} \to \mathrm{MRel}\\
\mathrm{MRel} &\triangleq \mathit{list}\ \mathit{Hom} \times \mathit{Mcont} \to \mathit{iProp} & \mathcal{K} &: \mathrm{VRel} \to \mathrm{VRel} \to \mathrm{CRel}\\
\mathrm{CRel} &\triangleq \mathit{Hom} \times \mathit{Ectx} \to \mathit{iProp} & \mathcal{E} &: \mathrm{VRel} \to \mathrm{VRel} \to \mathrm{VRel} \to \mathrm{ERel}\\
\mathrm{ERel} &\triangleq \mathbf{IT} \times \mathit{Expr} \to \mathit{iProp} & [\![\tau]\!] &: \mathrm{VRel}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{O}\big((\alpha,\kappa,\sigma),(e,K,mk)\big) &\triangleq \mathsf{has_state}(\sigma) \mathrel{-\!\!*} \\
&\quad \mathrm{wp}\,\mathbf{P}(\kappa\ \alpha)\ \{\beta.\ \exists v.\ (\langle e,\ K,\ mk\rangle_{\mathrm{eval}} \mapsto^{*} \langle v\rangle_{\mathrm{ret}}) \\
&\quad *(\beta,v) \in [\![\mathbb{N}]\!] * \mathsf{has_state}([])\}\\
\mathcal{M}(P)(\sigma,mk) &\triangleq \forall(\alpha,v).\ P(\alpha,v) \mathrel{-\!\!*} \mathcal{O}((\alpha,\iota,\sigma),(v,\Box,mk))\\
\mathcal{K}(Q,P)(\kappa,K) &\triangleq \Box\,\forall(\alpha,v).Q(\alpha,v) \mathrel{-\!\!*} \forall(\sigma,mk).\mathcal{M}(P)(\sigma,mk) \mathrel{-\!\!*} \\
&\quad \mathcal{O}((\alpha,\kappa,\sigma),(v,K,mk))\\
\mathcal{E}(P,Q,R)(\beta,e) &\triangleq \forall(\kappa,K).\mathcal{K}(P,Q)(\kappa,K) \mathrel{-\!\!*} \forall(\sigma,mk).\ \mathcal{M}(R)(\sigma,mk) \mathrel{-\!\!*} \\
&\quad \mathcal{O}((\beta,\kappa,\sigma),(e,K,mk))\\
[\![\mathbb{N}]\!](\beta,v) &\triangleq \exists n \in \mathbb{N}.\ \beta = \mathsf{Ret}(n) \wedge v = n\\
[\![\tau/\alpha \to \sigma/\beta]\!](\theta,v) &\triangleq \exists F.\ \theta = \mathsf{Fun}(F) \wedge \Box\,\forall(\eta,w).\ [\![\tau]\!](\eta,w) \mathrel{-\!\!*} \\
&\quad \mathcal{E}([\![\sigma]\!],[\![\alpha]\!],[\![\beta]\!])(\theta \bullet \eta, v\ w)\\
[\![\mathit{cont}(\tau,\alpha)]\!](\beta,v) &\triangleq \exists\kappa\ K.\ \beta = \mathsf{Fun}(\mathsf{next}(\lambda x.\ \mathsf{Tick}((\mathbf{P}\circ\kappa)\ x))) \wedge v = \mathsf{cont}\ K \wedge \\
&\quad \mathcal{K}([\![\tau]\!],[\![\alpha]\!])(\kappa,K)\\
[\![\Gamma]\!](\rho,\gamma) &\triangleq \forall(x:\tau\in\Gamma).\ \Box\,\forall\Phi.\ \mathcal{E}([\![\tau]\!],\Phi,\Phi)(\rho(x),\gamma(x))\\
\Gamma \vDash_{\mathsf{pure}} e:\tau &\triangleq \Box\,\forall(\rho,\gamma).\ [\![\Gamma]\!](\rho,\gamma) \mathrel{-\!\!*} \forall\Phi.\ \mathcal{E}([\![\tau]\!],\Phi,\Phi)(\mathbf{E}[\![e]\!]_{\rho}, e[\gamma])\\
\Gamma;\alpha \vDash e:\tau;\beta &\triangleq \Box\,\forall(\rho,\gamma).\ [\![\Gamma]\!](\rho,\gamma) \mathrel{-\!\!*} \mathcal{E}([\![\tau]\!],[\![\alpha]\!],[\![\beta]\!])(\mathbf{E}[\![e]\!]_{\rho}, e[\gamma])
\end{aligned}
$$

Fig. 23. Logical relation for λ_{delim}.

Proof of Theorem 5.2. Note that the empty (meta)continuation is related to its denotation: $\mathcal{K}(P,P)(\mathrm{id},\Box)$ and $\mathcal{M}(P)([],[])$ hold for any relation P.

With this, we instantiate $\emptyset;\mathbb{N} \vDash e:\mathbb{N};\mathbb{N}$ (that we get from Lemma 5.3) with the empty continuation/metacontinuation, and get the observational refinement between e and $\mathbf{E}[\![e]\!]$. □

This completes our treatment of denotational semantics of λ_{delim}. The next section examines interoperability of delimited continuations and other effects.

6 MODELING INTEROPERABILITY BETWEEN LANGUAGES

A key advantage of using (G)Itrees for semantics is that they can provide a common framework for multi-language interaction. This section presents a case study on the interaction between the languages λ_{embed} (with higher-order store effects) and λ_{delim} (with delimited continuations). Specifically, we allow embedding closed λ_{delim} programs into λ_{embed}, and equip λ_{embed} with a type system that guarantees safe interoperability.

The embedding we provide is restrictive, preventing programs with delimited continuations from accessing outer-language continuations. We leave developing a more permissive type system for future work. At the end of the section we give an example of how to verify a more involved interaction of effects, albeit without the type system.

$$\begin{array}{lll} \text{types} & Ty \ni \tau & ::= \mathbb{N} \mid \mathbf{1} \mid \tau \to \sigma \mid \mathrm{ref}(\tau) \\ \text{expressions} & Expr \ni e & ::= x \mid () \mid e_1\ e_2 \mid e_1 \oplus e_2 \mid n \mid \lambda x.\, e \\ & & \quad \mid \ell \mid \mathrm{ref}(e) \mid !\, e \mid e_1 \leftarrow e_2 \mid \mathsf{embed}\ e \end{array} \qquad \frac{\emptyset \vdash_{\mathsf{pure}} e : \mathbb{N}}{\Gamma \vdash \mathsf{embed}\ e : \mathbb{N}}$$

Fig. 24. Syntax and the new typing rule of the λ_{embed}.

In this section, we reuse the semantics of λ_{delim} from the previous section and higher-order store effects from Section 2. For λ_{delim}, we reuse the semantics from the previous section and higher-order store reifiers from [18]. In this section, we use magenta to explicitly highlight programs written in λ_{delim}, and for the interpretation functions of the denotational model of λ_{delim}.

Language λ_{embed}. λ_{embed} is a λ-calculus with base types $\mathbb{N}$ and $\mathbf{1}$, references types $\mathrm{ref}(\tau)$ and function types $\tau \to \sigma$, with syntax given in Figure 24. Additionally, it includes a construct $\mathsf{embed}\ e$ for embedding λ_{delim} programs. The typing rules are all standard, except for the new typing rule for the embedding.

The idea behind the rule is that we can embed an expression from λ_{delim} if it is a "pure" expression that can evaluate to a natural number. The use of pure typing judgment for the embedded program ensures that it does not to alter the answer type. This means that we can treat an embedded expression as a "complete" program, that does not require outer continuation delimiters, even though it may rely on delimited continuations internally. Those restrictions are crucial for the type safety of the embedding. The typing guarantees that e does not expect any additional delimiters, but it does not, by itself guarantee that any continuations in e escape the embedding boundary. To prevent that we enforce the continuation delimiter along the embedding boundary in the interpretation of embedded expressions.

Denotational model of λ_{embed}. For denotational semantics of λ_{embed}, we start by defining reifiers for the effect signature, which includes higher-order store operations (allocating, reading, and storing references) as $E_{\mathtt{state}}$, and effects related to delimited continuations ($E_{\lambda_{\mathsf{delim}}}$). Then the combined effect signature is $E_{\lambda_{\mathsf{delim}}} \times E_{\mathtt{state}}$, and thus we also let $State \triangleq State_{\mathtt{delim}} \times State_{\mathtt{state}}$, and reifiers are defined component-wise. Figure 25 shows the key parts of the denotational semantics. For most of the syntactic constructs we give the standard interpretation. For $\mathsf{embed}\ e$ we use the interpretation $\mathbf{E}[\![-]\!]$ for λ_{delim} from Section 5.2, and explicitly wrap the resulting GITree in a Reset. This continuation delimiter acts as a sort of *glue code* to protect the rest of the program from being captured by control operators from the embedded λ_{delim} program.

To show type safety of λ_{embed}, we construct a logical relation (shown in Figure 25), which is similar to the other logical relations we considered in this paper, mainly different in the observation relation O. Given that the type system for λ_{embed} effectively prevents expressions of λ_{delim} to access contexts from λ_{embed}, we refine the observation relation to get access to a version of the WP-HOM rule for expression interpretations of well-typed programs of λ_{embed}, which we do not have in general, as discussed in Section 3.

Throughout this paper, compositional reasoning has relied on WP-HOM: to reason about a computation running inside an evaluation context, we reason about the computation and the context separately. For context-dependent effects this rule is unsound in general, since an effect may inspect or capture the surrounding context (Section 3). For λ_{embed}, however, the type system prevents λ_{delim} expressions from accessing contexts that originate on the λ_{embed} side, so the control effects raised by a well-typed expression never escape into the ambient store context. A bind rule should therefore hold for well-typed expressions, even though we cannot derive it for the standard weakest precondition interpretation. To turn this intuition into a usable principle, we reason with a

$$\boxed{\mathbf{E}[\![-]\!] : \mathit{Expr} \to (\mathit{Var} \to \mathbf{IT}) \to \mathbf{IT}}$$

$$\mathbf{E}[\![x]\!]_\rho = \rho(x)$$
$$\mathbf{E}[\![n]\!]_\rho = \mathsf{Ret}(n)$$
$$\mathbf{E}[\![()]\!]_\rho = \mathsf{Ret}(())$$
$$\mathbf{E}[\![\ell]\!]_\rho = \mathsf{Ret}(\ell)$$
$$\mathbf{E}[\![\lambda x.\, e]\!]_\rho = \mathsf{Fun}(\mathsf{next}(\lambda x.\, \mathbf{E}[\![e]\!]_{\rho[x \mapsto x]}))$$
$$\mathbf{E}[\![\mathsf{embed}\ e]\!]_\rho = \mathsf{Reset}(\mathsf{next}(\mathbf{E}[\![e]\!]_\emptyset))$$

$$\mathbf{E}[\![e_1\ e_2]\!]_\rho = \mathsf{get_val}(\mathbf{E}[\![e_1]\!]_\rho, \lambda f.\ \mathsf{get_val}(\mathbf{E}[\![e_2]\!]_\rho, \lambda x. f \bullet x))$$
$$\mathbf{E}[\![e_1 \oplus e_2]\!]_\rho = \mathsf{NatOp}_\oplus(\mathbf{E}[\![e_1]\!]_\rho, \mathbf{E}[\![e_2]\!]_\rho)$$
$$\mathbf{E}[\![\mathsf{ref}(e)]\!]_\rho = \mathsf{get_val}(\mathbf{E}[\![e]\!]_\rho, \lambda x.\mathsf{Alloc}(x, \mathsf{Ret}))$$
$$\mathbf{E}[\![!\, e]\!]_\rho = \mathsf{get_val}(\mathbf{E}[\![e]\!]_\rho, \lambda x.\mathsf{Read}(x))$$
$$\mathbf{E}[\![e_1 \leftarrow e_2]\!]_\rho = \mathsf{get_val}(\mathbf{E}[\![e_2]\!]_\rho, \lambda x.\ \mathsf{get_ret}(\mathbf{E}[\![e_1]\!]_\rho, \lambda y.\mathsf{Write}(y, x)))$$

$$\mathsf{VRel} \triangleq \mathbf{IT}^v \to \mathit{iProp}$$
$$\mathsf{ERel} \triangleq \mathbf{IT} \to \mathit{iProp}$$
$$\mathcal{O} : \mathsf{VRel} \to \mathsf{ERel}$$
$$\mathcal{O}(P)(\beta) \triangleq \mathsf{clwp}\ \beta\ \{x.\ P\ x * \mathsf{has_state}([])\}$$
$$\mathcal{E} : \mathsf{VRel} \to \mathsf{ERel}$$
$$\mathcal{E}(P)(\beta) \triangleq \mathsf{heap_ctx} \mathrel{-\!\!*} \mathsf{has_state}([]) \mathrel{-\!\!*} \mathcal{O}(P)(\beta)$$

$$[\![\tau]\!] : \mathsf{VRel}$$
$$[\![\mathbf{1}]\!](\beta) \triangleq \beta = ()$$
$$[\![\mathbb{N}]\!](\beta) \triangleq \exists n.\, \beta = \mathsf{Ret}(n)$$
$$[\![\tau \to \sigma]\!](\beta) \triangleq \exists F.\, \beta = \mathsf{Fun}(F) \wedge \square\, \forall \beta.\, [\![\tau]\!](\beta) \mathrel{-\!\!*} \mathcal{E}([\![\sigma]\!])(\mathsf{Fun}(F) \bullet \beta)$$
$$[\![\mathsf{ref}(\tau)]\!](\beta) \triangleq \exists \ell.\, \beta = \mathsf{Ret}(\ell) \wedge \boxed{\exists v.\, \ell \mapsto v * [\![\tau]\!](v)}$$

$$[\![\Gamma]\!](\rho) \triangleq \forall (x : \tau \in \Gamma).\, \square\, \mathcal{E}([\![\tau]\!])(\gamma\ x)$$
$$\Gamma \vDash e : \tau \triangleq \forall \rho.\, [\![\Gamma]\!](\rho) \mathrel{-\!\!*} \mathcal{E}([\![\tau]\!])(\mathbf{E}[\![e]\!]_\rho)$$

Fig. 25. Denotational semantics and logical relation for λ_{embed}.

strengthened weakest precondition —*context-local weakest precondition* [36]—which bakes in the bind rule:

Definition 6.1. Context-local weakest precondition (clwp) is defined as follows: $\mathsf{clwp}\ \alpha\ \{\Phi\} \triangleq \forall \kappa\ (\Psi : \mathbf{IT}^v \to \mathit{iProp}).\, (\forall v.\, \Phi\ v \mathrel{-\!\!*} \mathsf{wp}\ (\kappa\ v)\ \{\Psi\}) \to \mathsf{wp}\ (\kappa\ \alpha)\ \{\Psi\}$

The definition quantifies over every continuation κ and result predicate Ψ: $\mathsf{clwp}\ \alpha\ \{\Phi\}$ holds when, in an arbitrary context κ that is itself safe on the values satisfying Φ, the computation $\kappa\ \alpha$ is safe with respect to Ψ. Intuitively, α is well-behaved in *any* well-behaved context. Quantifying over κ is precisely what bakes the bind rule into the definition, recovering for the well-typed fragment the context-local reasoning of Timany and Birkedal [36]. It also makes clwp stronger than wp: taking κ to be the identity and Ψ to be Φ shows that $\mathsf{clwp}\ \alpha\ \{\Phi\}$ implies $\mathsf{wp}\ \alpha\ \{\Phi\}$, while the converse fails in general. We thus use clwp only where the typing discipline of λ_{embed} lets us establish it, gaining compositional reasoning in return. In particular, clwp validates a form of the bind rule:

$$\mathsf{clwp}\ \alpha\ \{\beta.\ \mathsf{clwp}\ (\kappa\ \beta)\ \{\Phi\}\} \vdash \mathsf{clwp}\ (\kappa\ \alpha)\ \{\Phi\}$$

for any homomorphism κ. We use the clwp in the definition of observational refinement in the model. Observational refinement asserts that if the evaluation of a top-level expression of λ_{embed} starts with an empty continuation stack, then the evaluation does not introduce new elements into the continuation stack. By using clwp we get a semantical bind lemma, which can be seen as a version of WP-HOM for semantically valid expressions of λ_{embed}. As before, we then obtain fundamental lemma and denotational type soundness.

Lemma 6.2. *Semantical bind.* $\mathcal{E}(\lambda x : \mathbf{IT}^v.\, \mathcal{E}(P)(\kappa\ x))(\beta)$ *implies* $\mathcal{E}(P)(\kappa\ \beta)$ *for any homomorphism* κ.

$$
\begin{array}{ll}
\mathtt{prog} \triangleq \mathsf{Fun}(\mathsf{next}(\lambda y. & \\
\quad \mathsf{Let}\ x = \mathsf{Alloc}(\mathsf{Ret}(1))\ \mathsf{in} & \text{Initial offset value} \\
\quad \mathsf{Let}\ n = \mathsf{Shift}(\lambda k.\mathsf{next}(& \text{Capture continuation as } k \\
\qquad \mathsf{Appcont}'(\mathsf{Read}(x), k)\ ; & \text{First call to } k \text{ with the init. value of } x \\
\qquad \mathsf{Let}\ m = & x := x + 1; \\
\qquad\quad \mathsf{NatOp}_+(\mathsf{Read}(x), \mathsf{Ret}(1))\ \mathsf{in}\ \mathsf{Write}(x, m)\ ; & \\
\qquad \mathsf{Appcont}'(\mathsf{Read}(x), k))) & \text{Second call to } k \text{ with updated } x \\
\qquad \mathsf{in}\ \mathsf{get_ret}(y, (\lambda l.\mathsf{Let}\ p = & \\
\qquad\quad \mathsf{NatOp}_+(\mathsf{Read}(l), n)\ \mathsf{in}\ \mathsf{Write}(l, p))))) & y := y + n; \\
\qquad \text{where } \mathsf{Appcont}'(x, y) \triangleq \mathsf{Appcont}(\mathsf{next}(x), \mathsf{next}(\mathsf{Tau} \circ y \circ \mathsf{next})) &
\end{array}
$$

$$
\frac{\mathsf{heap_ctx} \qquad \mathsf{has_state}(\sigma) \qquad y \mapsto \mathsf{Ret}(n)}{\mathsf{wp}\ (\mathsf{Reset}(\mathsf{next}(\mathtt{prog} \bullet \mathsf{Ret}(y))))\ \{y \mapsto \mathsf{Ret}(n+3) * \mathsf{has_state}(\sigma)\}}
$$

Fig. 26. Example program with delimited continuations and state and its specification.

Lemma 6.3. *Fundamental lemma. Let* $\Gamma \vdash e : \tau$ *then* $\Gamma \vDash e : \tau$.

Lemma 6.4. *Denotational type soundness. Let* $\emptyset \vdash e : \tau$ *and* $(\mathbf{E}[\![e]\!]_\emptyset, [\,]) \rightsquigarrow^* (\alpha, \sigma)$, *then* $(\exists \beta\ \sigma'.\ (\alpha, \sigma) \rightsquigarrow (\beta, \sigma')) \vee (\alpha \in \mathbf{IT}^v)$.

Unrestricted interaction of delimited continuations and higher-order state. Even though the type system we considered here is restrictive, we can still reason about unrestricted interactions of events in the "untyped" setting. Here we show an example of such an unrestricted interaction, and demonstrate how to reason about context-dependent and context-independent effects at the same time. While this kind of interactions is forbidden by our type system, we can still write and prove meaningful specifications for such programs. Consider the program in Figure 26, written in GITrees directly. The function prog utilizes both delimited continuations and state. It takes a reference y as its argument and begins by allocating the value 1 in the store at reference x. Then it captures the continuation $\mathsf{get_ret}(y, (\lambda l.\mathsf{Let}\ p = \mathsf{NatOp}_+(\mathsf{Read}(l), \dots)\ \mathsf{in}\ \mathsf{Write}(l, p)))$ as k. Invoking the continuation k with a number n increments the current value of y by n. The program then invokes this continuation twice. First with the original value of x. Then, with an incremented value of x. Since the starting value of x is 1, the reference y is incremented first by 1 and then by 2. We capture this behavior in the specification for prog stated in Figure 26.

It is important to note that this program features a bidirectional interaction between state and continuations. Specifically, the body of Shift involves state operations, while the result of Shift is subsequently used to increment a value in the heap. As we have seen, while this type of interaction is not allowed in the type system, we can still reason about them in program logic. We stipulate that our proposed type system could potentially be extended to support embedding "pure" functions, allowing for bi-directional interaction between the two languages. We believe that such an extension would require implementing answer-type polymorphism, following the approach of Asai and Kameyama [2].

7 PREEMPTIVE CONCURRENCY

In this section we present yet another extension to GITrees: support for preemptive concurrency. This extension is orthogonal to the extension for context-dependent effects presented earlier, and demonstrates the flexibility of the GITrees framework. While concurrency does not involve context-dependent reification, it does require modifications to the reductions and the logic.

We proceed as follows. We first lift the operational machinery of GITrees to *threadpools*, modifying the step relation and the reifier signature so that an effect may spawn new threads. We then introduce a concurrent affine calculus, obtained by extending the affine language $\lambda_{\multimap,\mathsf{ref}}$ of Frumin et al. [18] with a fork concurrency construct. We give this calculus a denotational semantics into GITrees, and finally prove a semantic type safety theorem using the Iris-based program logic (Section 7.5). Type safety here is the property that a well-typed program never incurs a runtime error. What could go wrong, absent the affine discipline, is a violation of affine usage, *e.g.* a reference is used in one thread, but de-allocated in the other. The runtime affinity check guards against this, and proving type safety amounts to showing that the check never fails for well-typed programs, even under thread interleaving.

Since those checks are now carried out under thread interleavings, the checks themselves have to be thread-safe. For this reason we implement them with atomic operations. Implementation of atomic operations highlights a distinctive feature of GITrees: because effects can take *functions* as inputs, we define one generic atomic state-modification effect and recover specific ones as instances.

7.1 Threadpool Semantics

To support concurrency, we lift the step relation $\rightsquigarrow$ from individual GITrees to *threadpools*. A threadpool is simply a list of GITrees, representing threads running in parallel. The step relation for threadpools is defined as:

$$(\vec{\alpha}, \sigma) \rightsquigarrow_{\mathsf{tp}} (\vec{\alpha}', \sigma') \triangleq \exists i\, \alpha_i\, \alpha_i'.\, \vec{\alpha}[i] = \alpha_i \wedge (\alpha_i, \sigma) \rightsquigarrow (\alpha_i', \sigma', \vec{l}) \wedge \vec{\alpha}' = \vec{\alpha}[i \mapsto \alpha_i'] + \vec{l}$$

That is, one thread in the pool takes a step, updating both the thread and the global state. The other threads remain unchanged. Accordingly, we also modify signature of reifiers and allow $\rightsquigarrow$ to take forked threads into the account, as we describe in the next section.

The adequacy theorem (Theorem 2.3) extends naturally to threadpool reductions.

7.2 Fork Effect and Affine Language Extension

To spawn new threads we introduce a fork effect whose signature is shown in Figure 27. The effect takes as input a GITree e to be executed in a new thread, and returns unit. The reifier for fork, denoted $r_{\mathtt{fork}}$, takes the input GITree e, leaves the state σ unchanged, returns unit to the continuation κ, and produces a list containing the new thread e. This list is used to extend the threadpool with the newly spawned thread.

More precisely, the reifier signature is modified to return an optional triple $(\beta, \sigma', \vec{\alpha})$ where $\vec{\alpha}$ is a list of new threads to be added to the threadpool. For effects other than fork this list is empty.

This modification to the reifier signature is analogous to the extension for context-dependent effects presented earlier in this work. Just as context-dependent effects required reifiers to take the current continuation as an additional parameter (to allow effects like `call/cc` to capture and manipulate continuations), concurrency requires reifiers to return a list of new threads. Both extensions maintain backwards compatibility: effects that do not need these capabilities simply ignore the continuation parameter (for context-independent effects), or return an empty thread list (for non-forking effects). This demonstrates the flexibility of the GITrees framework: the reifier mechanism can be extended to accommodate different classes of effects while preserving the modularity of effect composition.

We demonstrate the fork effect by extending the affine language $\lambda_{\multimap,\mathsf{ref}}$ with a fork construct. The syntax and type system are shown in Figure 28. The expression $\mathsf{fork}(e_1, e_2)$ first evaluates e_1 (which typically performs some side effect), spawns it in a new thread, and then proceeds with e_2. The type system requires e_1 to have type Unit. The forked thread runs independently and is never joined, so its result is not used.

$$r : \prod_{\mathrm{i} \in E} \mathit{Ins}_{\mathrm{i}}(\mathbf{IT}_E) \times \mathit{State} \times (\mathit{Outs}_{\mathrm{i}}(\mathbf{IT}_E) \to \blacktriangleright \mathbf{IT}_E) \to \mathit{option}(\blacktriangleright \mathbf{IT}_E \times \mathit{State} \times \mathsf{list}(\blacktriangleright \mathbf{IT}_E))$$

$$\mathit{Ins}_{\mathsf{fork}}(X) \triangleq \blacktriangleright X \qquad \mathit{Outs}_{\mathsf{fork}}(X) \triangleq 1$$

$$r_{\mathsf{fork}}(e, \sigma, \kappa) = \mathsf{Some}(\kappa\ (), \sigma, [e])$$
$$\mathsf{Fork}(e) \triangleq \mathsf{Vis}_{\mathsf{fork}}(\mathsf{next}(e), \lambda x.\mathsf{next}(\mathsf{Ret}(x)))$$

Fig. 27. Modified reifier signature and fork effect.

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

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

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

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

Fig. 28. Type system for $\lambda_{\multimap,\mathsf{ref}}$ with concurrency.

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

7.3 Atomic State Modification via Lambda Functions in Effects

The interpretation of $\lambda_{\multimap,\mathsf{ref}}$ shown in Figure 29 shows an important aspect of concurrent programming with affine types. Consider the interpretation of variables: $[\![a]\!]_\rho \triangleq \mathsf{Force}(\rho(a))$, where:

$$\mathsf{Thunk}(\alpha) \triangleq \mathsf{Alloc}\big(\mathsf{Ret}(0), \lambda \ell.\, \mathsf{Fun}(\mathsf{next}(\lambda_.\, \mathsf{If}(\mathsf{Xchg}(\ell, \mathsf{Ret}(1)), \mathsf{Err}(\mathit{Lin}), \alpha)))\big)$$
$$\mathsf{Force}(\alpha) \triangleq \alpha \bullet \mathsf{Ret}(0)$$

The Thunk operation allocates a reference initialized to 0 (representing "not yet used"), and returns a thunk that, when forced, atomically exchanges the location's value with 1. If the exchange returns 0, the variable has not been used before, and we proceed with α. If it returns 1, the variable has already been used, violating the affine discipline, and we raise a linearity error.

Generic Atomic State Modification. The key technical challenge when extending GITrees with concurrency is handling shared mutable state. In a sequential setting the state effects Read and Write are enough, and the affine type system keeps references from being shared, so well-typed programs never race on ordinary state in the first place. The difficulty is more subtle: since we establish *semantic* type safety, the affine discipline must also be enforced *at runtime*, so that safety

holds even for the untyped programs the type system does not constrain, and this enforcement reads and updates a shared per-variable flag (as above). Carried out as a read of the flag followed by a separate write, this check is unproblematic sequentially; under thread interleaving, however, the same read-then-write races, since two threads could both read the flag as not used *before either marks it used*, and both would then proceed. One could address this in the program logic, *e.g.* by giving references a different logical representation; instead, we change the semantics of the check, requiring the flag to be read and updated in a single atomic step.

For this, we make use of the generic state modification effect as follows, which takes a modification function as input:

$$\begin{aligned} Ins_{\mathtt{atomic}}(X) &\triangleq Loc \times \blacktriangleright(X \to X \times X) \\ Outs_{\mathtt{atomic}}(X) &\triangleq \blacktriangleright X \end{aligned}$$

$$r_{\mathtt{atomic}}((\ell, f), \sigma, \kappa) = \begin{cases} \mathsf{Some}(\kappa\ (\mathsf{next}(r)), \sigma[\ell \mapsto \mathsf{next}(v')]), & \sigma[\ell] = \mathsf{Some}(v), f(v) = \mathsf{next}(r, v') \\ \mathsf{None}, & \text{otherwise} \end{cases}$$

This effect takes a location ℓ and a function $f : \blacktriangleright(X \to X \times X)$. The reifier atomically reads the current value $v = \sigma(\ell)$, applies f to it to obtain a pair (r, v'), stores v' back in ℓ, and returns r to the continuation. This design allows the function to compute both what to return and what to write, enabling a wide variety of atomic operations.

From this generic effect, we can derive various atomic operations. Atomic exchange is defined as:

$$\mathsf{Xchg}(\ell, w) \triangleq \mathsf{Vis}_{\mathtt{atomic}}((\ell, \lambda x.\ (\mathsf{next}(x), \mathsf{next}(w))), \mathsf{id})$$

The function $\lambda x.\ (\mathsf{next}(x), \mathsf{next}(w))$ returns the old value x as the result and the new value w to be written. Compare-and-swap (CAS) can be defined as:

$$\mathsf{CAS}(\ell, \mathit{exp}, \mathit{new}) \triangleq \mathsf{Vis}_{\mathtt{atomic}}((\ell, \lambda x.\ \mathsf{If}(x = \mathit{exp}, (\mathsf{Ret}(\mathit{true}), \mathsf{next}(\mathit{new})), (\mathsf{Ret}(\mathit{false}), \mathsf{next}(x)))), \mathsf{id})$$

which returns true and updates if the expected value matches, otherwise returns false and leaves the value unchanged. Fetch-and-add (FAA) is defined as:

$$\mathsf{FAA}(\ell, n) \triangleq \mathsf{Vis}_{\mathtt{atomic}}((\ell, \lambda x.\ (\mathsf{next}(x), \mathsf{next}(\mathsf{NatOp}_{+}(\mathsf{next}(x), \mathsf{next}(n))))), \mathsf{id})$$

which returns the old value and stores the incremented value.

Reasoning about Atomic Operations. The program logic rule for the atomic effect captures its semantics of both reading from, and writing to the state at the same time:

$$\textsc{wp-atomic}\ \frac{\begin{array}{c}\mathsf{heap_ctx} \\ \triangleright\, \ell \mapsto \alpha \\ \triangleright(f\ \alpha \equiv \mathsf{next}(w_1, w_2)) \\ \triangleright(\ell \mapsto w_2 \mathrel{-\!\!*} \Phi(w_1))\end{array}}{\mathsf{wp}\ \mathsf{Vis}_{\mathtt{atomic}}((\ell, f), \mathsf{id})\ \{\Phi\}}$$

Given ownership of location ℓ pointing to value α (written $\ell \mapsto \alpha$), and knowing that applying function f to α yields the pair (w_1, w_2), the rule allows us to conclude that after executing the atomic operation, we regain ownership of ℓ now pointing to w_2, and the result w_1 satisfies the postcondition. This rule is atomic in the sense that the read-modify-write sequence happens in a single logical step, which is essential for reasoning about concurrent programs where intermediate states must not be observable by other threads.

This demonstrates a unique capability of GITrees: by allowing effects to take functions as inputs, we can express atomic state modification generically, and derive various atomic primitives as needed. This is more flexible than hardcoding specific atomic operations into the effect signature.

$$
\begin{aligned}
[\![b]\!]_\rho &\triangleq \begin{cases} \mathsf{Ret}(1) & \text{if b = true} \\ \mathsf{Ret}(0) & \text{otherwise} \end{cases} \\
[\![()]\!]_\rho &\triangleq \mathsf{Ret}() \\
[\![n]\!]_\rho &\triangleq \mathsf{Ret}(n) \\
[\![a]\!]_\rho &\triangleq \mathsf{Force}(\rho(a)) \\
[\![\lambda a.\, e]\!]_\rho &\triangleq \mathsf{Fun}(\mathsf{next}(\lambda\alpha.\, [\![e]\!]_{\rho[a\mapsto\alpha]})) \\
[\![e_1\ e_2]\!]_\rho &\triangleq \mathsf{Let}\ x = [\![e_2]\!]_{\rho_2}\ \mathsf{in}\ [\![e_1]\!]_{\rho_1} \bullet \mathsf{Thunk}(x) \\
[\![(e_1,e_2)]\!]_\rho &\triangleq ([\![e_1]\!]_{\rho_2}, [\![e_2]\!]_{\rho_2}) \\
[\![\mathsf{let}\ (a_1,a_2) = e_1\ \mathsf{in}\ e_2]\!]_\rho &\triangleq \mathsf{Let}\ x = [\![e_1]\!]_{\rho_1}\ \mathsf{in}\ \mathsf{Let}\ y = \mathsf{Thunk}(\pi_1\ (x))\ \mathsf{in}\ \mathsf{Let}\ z = \mathsf{Thunk}(\pi_2\ (x))\ \mathsf{in} \\
&\qquad [\![e_2]\!]_{\rho_2[a_1\mapsto y, a_2\mapsto z]} \\
[\![\mathsf{alloc}(e)]\!]_\rho &\triangleq \mathsf{Let}\ x = [\![e]\!]_\rho\ \mathsf{in}\ \mathsf{Alloc}(x, \mathsf{Ret}) \\
[\![\mathsf{dealloc}(e)]\!]_\rho &\triangleq \mathsf{get_ret}([\![e]\!]_\rho, \mathsf{Dealloc}) \\
[\![\mathsf{replace}(e_1,e_2)]\!]_\rho &\triangleq \mathsf{Let}\ y = [\![e_2]\!]_{\rho_2}\ \mathsf{in}\ \mathsf{get_ret}([\![e_1]\!]_{\rho_1}, \lambda\ell.\, \mathsf{Let}\ x = \mathsf{Xchg}(\ell, y)\ \mathsf{in}\ (x, \mathsf{Ret}(\ell))) \\
[\![\mathsf{fork}(e_1,e_2)]\!]_\rho &\triangleq \mathsf{Fork}([\![e_1]\!]_{\rho_1})\ ;\ [\![e_2]\!]_{\rho_2}
\end{aligned}
$$

Fig. 29. Interpretation of $\lambda_{\multimap,\mathsf{ref}}$ with concurrency.

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

Fig. 30. Logical relation for $\lambda_{\multimap,\mathsf{ref}}$ with concurrency.

7.4 Interpretation of $\lambda_{\multimap,\mathsf{ref}}$ with Concurrency

The interpretation of $\lambda_{\multimap,\mathsf{ref}}$ expressions is shown in Figure 29. Most clauses are standard, but there are two important differences from the sequential version. First, the interpretation of $\mathsf{replace}(e_1, e_2)$ now uses Xchg instead of sequential Read and Write operations, reflecting the new implementation of Thunk. Second, the interpretation of $\mathsf{fork}(e_1, e_2)$ uses the Fork effect to spawn e_1 in a new thread before proceeding with e_2.

7.5 Program Logic and Type Safety

In this subsection we show how to treat the introduced effects in the program logic, and use it to construct a logical relation for proving type safety.

Once we treat the `fork` and `heap` effects in the program logic, we obtain a program logic that is essentially the same as the (standard) program logic of Iris. The main difference being that the Iris program logic is defined over concurrent programs, and ours is defined over GITrees. Everything else, including Iris tools like *invariants* and *ghost state*, is shared between the two. In particular

that means that we can, for free, support a variety of reasoning principles for concurrency; see Section 8 for references on how established methods can be encoded in Iris.

We now discuss the new weakest precondition calculus for GITrees with concurrency. First, we define two *invariants* that will allow us to share the effects state:

$$\mathsf{heapctx} \triangleq \boxed{\exists \sigma.\, \mathsf{has_state}_{\mathsf{heap}}(\sigma) * \bullet\, \sigma}^{\mathcal{N}_1}$$

$$\mathsf{forkctx} \triangleq \boxed{\mathsf{has_state}_{\mathsf{fork}}(())}^{\mathcal{N}_2}$$

These invariants ensure that the state is always owned by exactly one thread at a time, and can be temporarily acquired when performing an effectful operation.

The weakest precondition rules for the concurrent effects are shown below. The rule for Xchg requires ownership of the location $\ell \mapsto \alpha$, and after the exchange, the ownership is updated to $\ell \mapsto w$. Crucially, this rule is atomic: the read and write happen in a single step, preventing race conditions.

The rule for Fork requires the invariant forkctx and ensures that the forked thread α is safe (i.e., $\mathsf{wp}\ \alpha\ \{\top\}$). After forking, the parent thread continues with $\Phi(\mathsf{Ret}())$.

$$\frac{\mathsf{heapctx} \qquad \triangleright\, \ell \mapsto \alpha \qquad \triangleright(\ell \mapsto w \mathrel{-\!\!*} \mathsf{wp}\ \alpha\ \{\Phi\})}{\mathsf{wp}\ (\mathsf{Xchg}(\ell, w))\ \{\Phi\}} \qquad \frac{\mathsf{forkctx} \qquad \triangleright\, \mathsf{wp}\ \alpha\ \{\top\} \qquad \triangleright\, \Phi(\mathsf{Ret}())}{\mathsf{wp}\ (\mathsf{Fork}(\alpha))\ \{\Phi\}}$$

Using these rules, we verify the interpretation of $\lambda_{\multimap,\mathsf{ref}}$ shown in Figure 29. The logical relation in Figure 30 extends the standard logical relation for affine types with support for references and concurrency. In particular, the interpretation of reference types $\mathcal{V}[\![\mathsf{ref}\ \tau]\!](\beta_v)$ requires that the value is a location ℓ pointing to a value of type τ.

The semantic typing judgment $\Omega \models \alpha : \tau$ states that the interpretation α of an expression is safe and produces a value of type τ when executed in an environment satisfying Ω. Using this logical relation, we prove type soundness for $\lambda_{\multimap,\mathsf{ref}}$ with concurrency:

Lemma 7.1 (Fundamental Lemma for $\lambda_{\multimap,\mathsf{ref}}$). *If $\Omega \vdash e : \tau$, then $\Omega \models [\![e]\!] : \tau$.*

The type safety theorem for concurrent GITrees ensures that well-typed programs are safe and make progress:

Theorem 7.2 (Type Safety for Concurrent $\lambda_{\multimap,\mathsf{ref}}$). *Let $\vdash e : \tau$. Then for any threadpool $\vec{\alpha}$ and state σ such that $([\![e]\!], \sigma) \rightsquigarrow^*_{\mathsf{tp}} (\vec{\alpha}, \sigma')$, all threads in $\vec{\alpha}$ are either values or can take a step.*

This demonstrates that the concurrency extension to GITrees is sound and can be used to verify language interactions of a language with concurrency, shared mutable state and affine types with other languages.

8 DISCUSSION AND RELATED WORK

We conclude the paper by discussing related work and future directions.

This paper extends GITrees to handle context-dependent effects, which allows us to model higher-order languages with control operators like call/cc (x. e) and $\mathcal{S}$ x. e. We provided step-by-step guidance on designing such effects through the example of exceptions. Additionally, we presented an orthogonal extension for preemptive concurrency, demonstrating how higher-order effects enable generic atomic operations. We showed this extension supports interoperability between languages with different context-dependent effects, while preserving reasoning about context-independent effects. Our approach leverages the native support for higher-order functions and effects in GITrees. This differs from the first-order effect representation of effects in ITrees [41], which would require explicitly first-order representation of functions and continuations, if we want

to model first-class continuation. Such a model would mix syntactic and semantic concerns, which is part of what we are trying to avoid by working with (G)ITrees.

Another difference with ITrees is our approach to reasoning. While ITrees use bisimulation-based equational theory, we follow GITrees in using tailored program logics and defining refinements. Our logic is expressive enough to define logical relations and carry out computational adequacy proofs. In future work, it would be interesting to develop techniques for reasoning about weaker notions of equality than the basic equational theory that GITrees comes equipped with (see the more extensive discussion of this point in [18]).

The "classical" domain theory remains an important source of inspiration and ideas for our development, and we want to mention some of the related work along those lines. Cartwright and Felleisen [11] introduced a framework of extensible direct models for constructing modular denotational semantics of programming languages. Their framework centers on an abstract notion of *resources* for representing effects (such as store or continuations) and a central `admin` function that manages these resources. Each language extension defines both the types (domains) of additional values and resources, and specifies the *actions* that the `admin` function can perform on these resources. Building on this framework, Sitaram and Felleisen [34] demonstrated that such models can provide direct-style fully abstract semantics for control operators. Their approach interprets effects, including continuations, by delegating them to a top-level handler. Our work adopts several key ideas from this line of research but reformulates them in the context of GITrees rather than classical domain theory. In our framework, effect signatures define resources, reifiers specify actions, and the reduction relation serves as the central authority dispatching the effects. The transition to GITrees enables us to formalize the extensibility of this approach in a practical manner, and it allows us to develop program logics where "resources" (as above) become resources in separation logic.

Compared to other programming languages paradigms and effects, type systems and logical relations for delimited continuations have not been studied as comprehensively. The original type system for `shift`/`reset` is due to Danvy and Filinski [12], where they employ *answer-type modification*. Materzok and Biernacki [27] generalized this type system to account for more involved control operators $\texttt{shift}_0$/$\texttt{reset}_0$; an alternative substructural type system for these operators was designed by Kiselyov and Shan [22]. Dyvbig, Peyton Jones, and Sabry [16] provide a typed monadic account of CPS for delimited continuations with dynamic prompt generation. Asai and Kameyama [2] present a polymorphic variant of the Danvy and Filinski's type system.

Biernacka and Biernacki [6] prove termination for a language with $\mathcal{S}$ x. e and $\mathcal{D}$ e (but without recursion) using logical relations based on abstract machine semantics. The shape of their logical relation is similar to our logical relation used for showing adequacy in that they also have relations for configurations, metacontinuations, *etc.* Asai [1] uses type-directed logical relations to verify a direct-style specializer (partial evaluator) for a language with $\mathcal{S}$ x. e and $\mathcal{D}$ e, proving correctness against evaluation-context based operational semantics. In contrast, our logical relation relates the GITree denotation of a program to its operational semantics, which we use to establish soundness. Moreover, it is a step-indexed relation defined within the Iris-based program logic for GITrees and fully mechanized in Rocq, rather than an on-paper, set-theoretic construction.

Reasoning about concurrency. Our program logic is an instance of the Iris program logic, which lets us reuse Iris's machinery for reasoning about concurrent programs. In particular, established methods such as Owicki-Gries [29] and rely/guarantee are available through the Views framework [15], which recovers them as instances of a common metatheory. Our contribution is not a new such method but lifting the GITrees framework to a concurrent setting, with thread-pool

semantics and a thread-spawning reifier, so that these principles apply to GITree denotations, as we do in Section 7.5.

Monadic accounts of control. Interpreting control by passing continuations is classical: Moggi [28] structures effects as monads, with the continuation monad capturing first-class control, and Atkey [3] generalizes this to parameterised monads modelling the answer-type modification behind `shift`/`reset`. Our context-dependent reifiers share this perspective. The contribution is not a new notion of computation, but a framework-level integration: generalizing the GITrees reifier so that one guarded datatype and its Iris program logic handle context-dependent effects, while still recovering the bind rule for effects that do not inspect the context.

In our interoperability example we showed type safety of a combined language, with respect to denotational semantics. In future work we would like to examine other properties: for example, that λ_{delim} expressions cannot disrupt λ_{embed}'s control flow, perhaps establishing some form of well-bracketedness as in [37].

We would also like to study other context-dependent effects like handlers and algebraic effects [4, 31, 32, 39, 40]. In particular, it would be interesting to give a denotational semantics to a language with handlers, derive program logic rules for it, and compare the resulting program logic to the one in [14]. Additionally, the concurrency extension could be developed further to handle more sophisticated synchronization primitives and scheduling policies.

ACKNOWLEDGMENTS

This work was co-funded by the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them. 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.

REFERENCES

[1] Kenichi Asai. 2005. Logical relations for call-by-value delimited continuations. In *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005 (Trends in Functional Programming, Vol. 6)*, Marko C. J. D. van Eekelen (Ed.). Intellect, 63–78.

[2] Kenichi Asai and Yukiyoshi Kameyama. 2007. Polymorphic Delimited Continuations. In *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4807)*, Zhong Shao (Ed.). Springer, 239–254. https://doi.org/10.1007/978-3-540-76637-7_16

[3] Robert Atkey. 2009. Parameterised notions of computation. *J. Funct. Program.* 19, 3-4 (2009), 335–376. https://doi.org/10.1017/S095679680900728X

[4] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.* 84, 1 (2015), 108–123. https://doi.org/10.1016/J.JLAMP.2014.02.001

[5] Nick Benton, Andrew Kennedy, and Carsten Varming. 2009. Some Domain Theory and Denotational Semantics in Coq. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 115–130. https://doi.org/10.1007/978-3-642-03359-9_10

[6] Malgorzata Biernacka and Dariusz Biernacki. 2009. Context-based proofs of termination for typed delimited-control operators. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, António Porto and Francisco Javier López-Fraguas (Eds.). ACM, 289–300. https://doi.org/10.1145/1599410.1599446

[7] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. 2005. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. *Log. Methods Comput. Sci.* 1, 2 (2005). https://doi.org/10.2168/LMCS-1(2:5)2005

[8] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting algebraic effects. *Proc. ACM Program. Lang.* 3, POPL, Article 6 (jan 2019), 28 pages. https://doi.org/10.1145/3290319

[9] 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

[10] Venanzio Capretta. 2005. General recursion via coinductive types. *Log. Methods Comput. Sci.* 1, 2 (2005). https://doi.org/10.2168/LMCS-1(2:1)2005

[11] Robert Cartwright and Matthias Felleisen. 1994. Extensible Denotational Language Specifications. In *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings (Lecture Notes in Computer Science, Vol. 789)*, Masami Hagiya and John C. Mitchell (Eds.). Springer, 244–272. https://doi.org/10.1007/3-540-57887-0_99

[12] Olivier Danvy and Andrzej Filinski. 1989. *A functional abstraction of typed contexts.* Technical Report. DIKU – Computer Science Department, University of Copenhagen.

[13] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, Gilles Kahn (Ed.). ACM, 151–160. https://doi.org/10.1145/91556.91622

[14] Paulo Emílio de Vilhena and François Pottier. 2021. A separation logic for effect handlers. *Proc. ACM Program. Lang.* 5, POPL, Article 33 (jan 2021), 28 pages. https://doi.org/10.1145/3434314

[15] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 287–300. https://doi.org/10.1145/2429069.2429104

[16] R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. 2007. A monadic framework for delimited continuations. *J. Funct. Program.* 17, 6 (2007), 687–730. https://doi.org/10.1017/S0956796807006259

[17] Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, Martin Wirsing (Ed.). North-Holland, 193–222.

[18] Dan Frumin, Amin Timany, and Lars Birkedal. 2024. Modular Denotational Semantics for Effects with Guarded Interaction Trees. *Proc. ACM Program. Lang.* 8, POPL (2024), 332–361. https://doi.org/10.1145/3632854

[19] Iris team. 2025. *The Iris Project website and Coq development.* https://iris-project.org/

[20] 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

[21] Alexis King. 2021. Delimited continuation primops. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0313-delimited-continuation-primops.rst. Accessed: 2024-06-27.

[22] Oleg Kiselyov and Chung-chieh Shan. 2007. A Substructural Type System for Delimited Continuations. In *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4583)*, Simona Ronchi Della Rocca (Ed.). Springer, 223–239. https://doi.org/10.1007/978-3-540-73228-0_17

[23] 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, Cascais, Portugal, January 14-15, 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM, 234–248. https://doi.org/10.1145/3293880.3294106

[24] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. 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

[25] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS, Vol. 153)*, Paul Blain Levy and Neel Krishnaswami (Eds.). 100–126. https://doi.org/10.4204/EPTCS.153.8

[26] Daan Leijen. 2017. Structured asynchrony with algebraic effects. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 3, 2017*, Sam Lindley and Brent A. Yorgey (Eds.). ACM, 16–29. https://doi.org/10.1145/3122975.3122977

[27] Marek Materzok and Dariusz Biernacki. 2011. Subtyping delimited continuations. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 81–93. https://doi.org/10.1145/2034773.2034786

[28] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[29] Susan S. Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM* 19, 5 (1976), 279–285. https://doi.org/10.1145/360051.360224

[30] Andrew M. Pitts. 2004. Typed Operational Reasoning. In *Advanced Topics In Types And Programming Languages*, Benjamin C. Pierce (Ed.). The MIT Press, Chapter 7, 245–289.

[31] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Log. Methods Comput. Sci.* 9, 4 (2013). https://doi.org/10.2168/LMCS-9(4:23)2013

[32] Casper Bach Poulsen and Cas van der Rest. 2023. Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects. *Proc. ACM Program. Lang.* 7, POPL (2023), 1801–1831. https://doi.org/10.1145/3571255

[33] Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. 2023. Semantics for Noninterference with Interaction Trees. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, Seattle, Washington, United States, July 17-21, 2023 (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:29. https://doi.org/10.4230/LIPICS.ECOOP.2023.29

[34] Dorai Sitaram and Matthias Felleisen. 1991. Modeling Continuations without Continuations. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, David S. Wise (Ed.). ACM Press, 185–196. https://doi.org/10.1145/99583.99611

[35] Sergei Stepanenko, Emma Nardino, Dan Frumin, Amin Timany, and Lars Birkedal. 2025. Context-Dependent Effects in Guarded Interaction Trees. In *Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 15695)*, Viktor Vafeiadis (Ed.). Springer, 286–313. https://doi.org/10.1007/978-3-031-91121-7_12

[36] Amin Timany and Lars Birkedal. 2019. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.* 3, ICFP (2019), 105:1–105:28. https://doi.org/10.1145/3341709

[37] Amin Timany, Armaël Guéneau, and Lars Birkedal. 2024. The Logical Essence of Well-Bracketed Control Flow. *Proc. ACM Program. Lang.* 8, POPL (2024), 575–603. https://doi.org/10.1145/3632862

[38] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6 (2024), 40:1–40:75. https://doi.org/10.1145/3676954

[39] 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

[40] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). ACM, 1–12. https://doi.org/10.1145/2633357.2633358

[41] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

[42] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473572

[43] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. 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, Rome, Italy (Virtual Conference), June 29 - July 1, 2021 (LIPIcs, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:19. https://doi.org/10.4230/LIPICS.ITP.2021.32