Context-Dependent Effects in Guarded Interaction Trees

Sergei Stepanenko[1], Emma Nardino[2], Dan Frumin[3], Amin Timany[1], and Lars Birkedal[1]

[1] Aarhus University, Aarhus, Denmark
`{sergei.stepanenko,timany,birkedal}@cs.au.dk`
[2] Univ Lyon, ENS de Lyon, UCBL, CNRS, Inria, LIP, UMR 5668, F-69342, Lyon CEDEX 07, France `emma.nardino@ens-lyon.fr`
[3] University of Groningen, Groningen, The Netherlands
`d.frumin@rug.nl`

Abstract. Guarded Interaction Trees are a structure and a fully formalized framework for representing higher-order computations with higher-order effects in Coq. 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so-called "bind rule" in modern program logics (which allows one to reason modularly about a term inside a context) is no longer valid.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 Coq 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.

Keywords: Coq · Iris · denotational semantics · logical relations · control flow operators · continuations · delimited continuations

1 Introduction

Despite a lot of recent progress, representing and reasoning about programming languages in proof assistants, such as Coq, is still considered a major challenge.

The design space is wide and many approaches have been considered. Recently, research on a novel point in the design space was initiated with the introduction of Interaction Trees [34], or ITrees for short. ITrees were introduced to simplify representation and reasoning about possibly non-terminating programs with side effects in Coq. 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., [18,36,35,25], 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 [13].

While GITrees support *higher-order effects*, in particular, the challenging case of higher-order store, they are limited to effects that do not alter the control flow of the program (more specifically, 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`, exceptions, or delimited continuations. 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 Coq, 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 [16]. Moreover, effect handlers, which rely on control flow operators, have recently been introduced in OCaml 5.0, and as a design feature in Helium [7], Koka [21,20], 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 technical development and key challenges. Similarly to how ITrees are defined as a coinductive type in Coq, GITrees are defined as a guarded recursive type (this is to support function spaces in GITrees; 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). Coq does not directly support guarded recursive types, so GITrees are defined using a fragment of guarded type theory implemented in Coq, as part of the Iris framework [14]. To work efficiently with GITrees we make use of other Iris features like separation logic and Iris Proof Mode [19], 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 Coq, in much the same way as one works directly in Coq. 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 (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 GITrees 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 $\lambda_{\mathsf{callcc}}$ with `call/cc` (Section 3). 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.

Our main interest, however, lies in the treatment of delimited continuations. In Section 4 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). 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 [30].

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 5) an example of such interaction: we show a type-safe embedding of λ_{delim} with delimited continuations into a language λ_{embed} 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 4) when proving type safety for the combined language. At the end of Section 5, we give an example of how to verify a more involved interaction of effects, albeit without the type system.

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 sound and adequate model of a calculus with delimited continuations, with operations `shift` and `reset`, implemented in a direct style (Section 4).
- 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 5).

All results in the paper have been formalized in Coq as a modification to the GITrees library and the previously proved results have been ported to our extension. We conclude and discuss related work in Section 6. Before we go on with the main part of the paper, we recall some background material on GITrees.

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 [13].

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 [15] and guarded type theory [8] 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 [19], implemented in Coq. This allows the users of Iris to carry out formal reasoning in separation logic as if they are proving things normally Coq, as we have done in the formalization of this work. For this reason, in the paper we will

$$\begin{aligned}
\tau &::= \mathsf{iProp} \mid 0 \mid \mathbf{1} \mid \mathbb{B} \mid \mathsf{Nat} \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau \mid \blacktriangleright\tau \mid I \mid \Sigma_{\mathtt{i}\in\mathtt{I}}\tau_{\mathtt{i}} \mid \Pi_{\mathtt{i}\in\mathtt{I}}\tau_{\mathtt{i}} \mid \ldots \\
t &::= x \mid F(t_1, \ldots, t_n) \mid \mathsf{abort}\ t \mid () \mid (t, t) \mid \pi_i\ t \mid \lambda x : \tau.\, t \mid \\
&\qquad \mathsf{inj}_i\ t \mid \mathsf{match}\ t\ \mathsf{with}\ \overline{\mathsf{inj}_i\ x.t}\ \mathsf{end} \mid \mathsf{next}(t) \mid \mathit{fix}_\tau \mid \ldots \\
P &::= \mathsf{False} \mid t =_\tau t \mid P \lor P \mid P \to P \mid \forall x : \tau.\, P \mid P * P \mid P \mathrel{-\!\!*} P \mid \Box P \mid \boxed{P} \mid \triangleright P \mid \ldots
\end{aligned}$$

Fig. 1. Grammar for the Iris base logic.

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

Fig. 2. Guarded datatype of interaction trees.

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. 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 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 $\mathbf{IT}^v_E(A)$ for the type of $\mathbf{IT}_E(A)$-values.

The nodes of the computation trees are of the 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:

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

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 $\mathsf{get\_val}(\alpha, f : \mathbf{IT}^v \to \mathbf{IT})$ are used for sequencing computations, and its corresponding equations are shown in Figure 3. Intuitively, $\mathsf{get\_val}(\alpha, f)$ 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, $\mathsf{get\_fun}(\alpha : \mathbf{IT}, f : \blacktriangleright(\mathbf{IT} \to \mathbf{IT}) \to \mathbf{IT})$ and $\mathsf{get\_ret}(\alpha : \mathbf{IT}_E(A), f : A \to \mathbf{IT}_E(A))$ 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.

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

Fig. 3. Example function on Guarded Interaction Trees.

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

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

Fig. 4. Signature of reifiers and the reification function

Crucially, to work with higher-order computations, GITrees provide the "call-by-value" application $\alpha \bullet \beta$ 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 $\mathsf{NatOp} : (\mathbb{N} \to \mathbb{N} \to \mathbb{N}) \to IT \to IT \to IT$ 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 $\blacktriangleright\mathbf{IT}$ (representing the heap); and we define the following reifier functions:

$$\begin{aligned} r_{\texttt{write}}((\ell,\alpha),\sigma) &= \mathsf{Some}((),\sigma[\ell\mapsto\alpha]) && \text{(where } \ell\in\sigma\text{, and } \mathsf{None} \text{ otherwise)}\\ r_{\texttt{read}}(\ell,\sigma) &= \mathsf{Some}(\alpha,\sigma) && \text{(where } \sigma(\ell)=\alpha\text{, and } \mathsf{None} \text{ otherwise)}\\ r_{\texttt{alloc}}(\alpha,\sigma) &= \mathsf{Some}(\ell,\sigma[\ell\mapsto\alpha]) && \text{(where } \ell\notin\sigma)\end{aligned}$$

Given reifiers for all the effects, we define a function $\mathsf{reify} : \mathbf{IT}\times \mathit{State} \to \mathbf{IT}\times \mathit{State}$ (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:

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

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 $\mathsf{Tau}\circ\mathsf{next}$.

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 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.

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 predicate Φ. 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 disentangle a homomorphism from the GITree it's applied to:

$$\textsc{wp-reify}\quad \frac{\begin{array}{c}\mathsf{has\_state}(\sigma)\\ \mathsf{reify}(\mathsf{Vis}_i(x,k),\sigma) = (\mathsf{Tick}(\beta),\sigma')\\ \triangleright\left(\mathsf{has\_state}(\sigma') \twoheadrightarrow \mathsf{wp}\,\beta\,\{\Phi\}\right)\end{array}}{\mathsf{wp}\;\mathsf{Vis}_i(x,k)\;\{\Phi\}} \qquad \textsc{wp-hom}\quad \frac{f \in \mathit{Hom} \quad \mathsf{wp}\,\alpha\,\{\beta_v.\,\mathsf{wp}\,f(\beta_v)\,\{\Phi\}\}}{\mathsf{wp}\,f(\alpha)\,\{\Phi\}}$$

Fig. 5. Selected weakest precondition rules.

Lemma 1. *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 1. *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 Coq system; thus, the adequacy theorem allows us to relate proofs inside the program logic (Iris) to the proofs on the level of Coq. This aspect is important in Iris and GITrees in general, but it is orthogonal to the work that we present in this paper. See [13] for more details.

3 Context-Dependent Reification

In this section we extend reification to handle context-dependent effects, using a language $\lambda_{\mathsf{callcc}}$ with `call/cc` as a concrete example. In Section 3.1 we present $\lambda_{\mathsf{callcc}}$'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 $\lambda_{\mathsf{callcc}}$ 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 $\lambda_{\mathsf{callcc}}$, 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.

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

$$
\mathsf{call/cc}\ (\mathsf{x}.\ e) \mapsto_K e[\mathsf{cont}\ K/x] \qquad K[\mathsf{throw}\ v\ \mathsf{to}\ \mathsf{cont}\ K'] \mapsto K'[v] \qquad \frac{e_1 \mapsto_K e_2}{K[e_1] \mapsto K[e_2]}
$$

$$
\frac{\Gamma, x : \mathit{cont}(\tau) \vdash e : \tau}{\Gamma \vdash \mathsf{call/cc}\ (\mathsf{x}.\ e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \mathit{cont}(\tau)}{\Gamma \vdash \mathsf{throw}\ e_1\ \mathsf{to}\ e_2 : \tau'}
$$

Fig. 6. Syntax and fragments of type system and operational semantics of $\lambda_{\mathsf{callcc}}$.

3.1 Operational Semantics and Type System for $\lambda_{\mathsf{callcc}}$

By $\lambda_{\mathsf{callcc}}$ 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 $\mathsf{call/cc}\ (\mathsf{x}.\ e)$ expression takes the current evaluation context and binds it to x in e. The $\mathsf{throw}\ e\ \mathsf{to}\ e'$ expression evaluates passes the first argument (value) to the second argument (continuation, represented as an evaluation context).

The operational semantics of $\lambda_{\mathsf{callcc}}$ 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 $\mathsf{call/cc}\ (\mathsf{x}.\ e)$ to capture the evaluation context. Let us now consider what happens if we try to give a direct-style denotational semantics of $\lambda_{\mathsf{callcc}}$ 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 a continuation, it is natural to let the input arity for `callcc` be a callback $(\blacktriangleright\mathbf{IT} \to \blacktriangleright\mathbf{IT}) \to \blacktriangleright\mathbf{IT}$. The output arity is simply $\blacktriangleright\mathbf{IT}$.

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.

$$
\begin{array}{rlrl}
\mathit{Ins}_{\texttt{callcc}}(X) & \triangleq ((\blacktriangleright X \to \blacktriangleright X) \to \blacktriangleright X) & \mathit{Outs}_{\texttt{callcc}}(X) & \triangleq \blacktriangleright X \\
\mathit{Ins}_{\texttt{throw}}(X) & \triangleq \blacktriangleright X \times \blacktriangleright(X \to X) & \mathit{Outs}_{\texttt{throw}}(X) & \triangleq 0 \\
\mathrm{Callcc}(f) & \triangleq \mathsf{Vis}_{\texttt{callcc}}(f, \mathsf{id}) & \mathrm{Throw}(e, f) & \triangleq \mathsf{Vis}_{\texttt{throw}}(e, f, \lambda x.\, \mathsf{abort}\ x)
\end{array}
$$

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

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

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

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

For convenience, we will use the abbreviations Callcc(f) and 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 $\mathbf{1}$ (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. 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 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: $\mathit{Outs}_i(\mathbf{IT}_E) \to \blacktriangleright \mathbf{IT}_E$. 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 is not supposed to be interrupted. For example, since the evaluation of a $\mathsf{call/cc}\ (\mathsf{x}.\ e)$ 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 $r_{\texttt{callcc}}(x, \sigma, \kappa) = \mathsf{Some}(\kappa\ (x\ \kappa), \sigma)$.

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

$$\textsc{wp-write}\quad\frac{\begin{array}{c}\mathsf{heap\_ctx}\quad \triangleright \ell \mapsto \alpha\\ \triangleright(\ell \mapsto \beta \mathrel{-\!\!*}\\ \mathsf{wp}\ \mathsf{Ret}()\ \{\Phi\})\end{array}}{\mathsf{wp}\ \mathsf{Write}(\ell,\beta)\ \{\Phi\}}
\qquad
\textsc{wp-write-ctx-dep}\quad\frac{\begin{array}{c}\kappa \in \mathit{Hom}\\ \mathsf{heap\_ctx}\quad \triangleright \ell \mapsto \alpha\\ \triangleright(\ell \mapsto \beta \mathrel{-\!\!*}\\ \mathsf{wp}\ \kappa\ (\mathsf{Ret}())\ \{\Phi\})\end{array}}{\mathsf{wp}\ \kappa\ (\mathsf{Write}(\ell,\beta))\ \{\Phi\}}
\qquad
\textsc{wp-reify-ctx-dep}\quad\frac{\begin{array}{c}\mathsf{has\_state}(\sigma)\\ r_i(x,\sigma,k) = \mathsf{Some}(\mathsf{next}(\beta),\sigma')\\ \triangleright\big(\mathsf{has\_state}(\sigma') \mathrel{-\!\!*}\\ \mathsf{wp}\ \beta\ \{\Phi\}\big)\end{array}}{\mathsf{wp}\ \mathsf{Vis}_i(x,k)\ \{\Phi\}}$$

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

dispatching them) parallels Cartwright and Felleisen's "extensible direct models" [9], which also aimed to support extensible denotational semantics in classical domain theory. We discuss this more in Section 6.

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 [28] in their development of a program logic for `call/cc`.) The reason is that context-dependent reification invalidates Lemma 1. Now, since WP-HOM is not sound anymore, one might expect that we need to adapt all the other program logic rules to include a homomorphism similarly to how the rules of [28] were adapted to include an evaluation context. However, this is not necessary, because our program logic is defined on *denotations* on which we have a non-trivial equational theory, which can be used to reason about 'pure' GITrees. Only for effectful operations, the proof rules will now have to include a surrounding homomorphism. E.g., WP-WRITE from [13] is generalized to WP-WRITE-CTX-DEP, and considers ambient homomorphisms explicitly.

Our context-dependent reification extension, though simple, allows us to build sound and adequate denotational models for languages with control-flow operators, including $\lambda_{\mathsf{callcc}}$ (shown in the next subsection). Moreover, our extension is conservative, and we recover previous case studies (computational adequacy of $\lambda_{\mathsf{rec,io}}$ and type safety for $\lambda_{\multimap,\mathsf{ref}}$ [13]) with minimal modifications; see the accompanying Coq 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 [13]), and validates the utility of the program logic.

$$\mathbf{E}[\![x]\!]_\rho = \rho(x)$$
$$\mathbf{E}[\![\textsf{call/cc}\ (\textsf{x}.\ e)]\!]_\rho = \mathrm{Callcc}(\lambda(f : \blacktriangleright\mathbf{IT} \to \blacktriangleright\mathbf{IT}).\ \mathbf{E}[\![e]\!]_{\rho[x\mapsto\mathsf{Fun}(\mathsf{next}(\lambda y.\ \mathsf{Tau}(f(\mathsf{next}(y)))))]})$$
$$\mathbf{E}[\![\textsf{throw}\ e_1\ \textsf{to}\ e_2]\!]_\rho = \mathsf{get\_val}(\mathbf{E}[\![e_1]\!]_\rho, \lambda x.\ \mathsf{get\_fun}(\mathbf{E}[\![e_2]\!]_\rho, \lambda f.\ \mathrm{Throw}(x, f)))$$
$$\mathbf{V}[\![\textsf{cont}\ K]\!]_\rho = \mathsf{Fun}(\mathsf{next}(\lambda x.\ \mathsf{Tau}(\mathbf{K}[\![K]\!]_\rho\ (\blacktriangleright\bullet)\ \mathsf{next}(x))))$$
$$\mathbf{K}[\![\textsf{throw}\ K\ \textsf{to}\ e]\!]_\rho = \lambda x.\ \mathsf{get\_val}(\mathbf{K}[\![K]\!]_\rho\ x, \lambda y.\ \mathsf{get\_fun}(\mathbf{E}[\![e]\!]_\rho, \lambda f.\ \mathrm{Throw}(y, f)))$$
$$\mathbf{K}[\![\textsf{throw}\ v\ \textsf{to}\ K]\!]_\rho = \lambda x.\ \mathsf{get\_val}(\mathbf{V}[\![v]\!]_\rho, \lambda y.\ \mathsf{get\_fun}(\mathbf{K}[\![K]\!]_\rho\ x, \lambda f.\ \mathrm{Throw}(y, f)))$$

Fig. 10. Denotational semantics of $\lambda_{\mathsf{callcc}}$ (selected clauses).

Interpretation of $\lambda_{\mathsf{callcc}}$. The denotational semantics of $\lambda_{\mathsf{callcc}}$ is shown in Figure 10 (selected clauses only; see Coq formalization for the complete definition). The interpretation is split into three parts: $\mathbf{E}[\![-]\!]$ for expressions, $\mathbf{V}[\![-]\!]$ for values, and $\mathbf{K}[\![-]\!]$ for contexts. For the interpretation of throw e_1 to e_2, the left-to-right evaluation order is enforced by the functions get_val and get_fun. They first evaluate their argument to a GITree value, and then pass it on (c.f. Figure 3).

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 2. *For any context K and an environment ρ, we have $\mathbf{K}[\![K]\!]_\rho \in$ 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. *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
$$\frac{\kappa \in \mathit{Hom} \quad \mathsf{has\_state}(\sigma) \qquad \triangleright(\mathsf{has\_state}(\sigma) -\!\!\ast\ \mathsf{wp}\ f\ x\ \{\Phi\})}{\mathsf{wp}\ \kappa\ (\mathrm{Throw}(\mathsf{next}(x), \mathsf{next}(f)))\ \{\Phi\}}$$

WP-CALLCC
$$\frac{\kappa \in \mathit{Hom} \quad \mathsf{has\_state}(\sigma) \qquad \triangleright(\mathsf{has\_state}(\sigma) -\!\!\ast\ \mathsf{wp}\ \kappa\ (f\ \kappa)\ \{\Phi\})}{\mathsf{wp}\ \kappa\ (\mathrm{Callcc}(\mathsf{next} \circ f))\ \{\Phi\}}$$

where κ is a homomorphism representing the current evaluation context on the level of GITrees. The reader may wonder why these rules include the has_state(σ) 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.

$$\mathcal{O}(\alpha, e) \triangleq \mathsf{has\_state}(()) \twoheadrightarrow \mathsf{wp}\ \alpha \left\{\beta.\exists v.\, (e \mapsto^* v) * [\![\mathbb{N}]\!](\beta, v) * \mathsf{has\_state}(())\right\}$$

$$\boxed{\mathcal{O} : \mathrm{ERel}}$$

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

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

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

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

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

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

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

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

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

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

Lemma 4. *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 4, 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 [23], adapted from [28] for adequacy, following the Iris approach [30].

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

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

Lemma 6. *Fundamental lemma.* Let $\Gamma \vdash e : \tau$ then $\Gamma \vDash e : \tau$.

$$\boxed{\begin{array}{l}\Gamma;\alpha \vdash e : \tau;\beta \\ \Gamma \vdash_{\mathsf{pure}} e : \tau\end{array}}$$

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

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

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

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

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

Fig. 12. Syntax and typing rules of λ_{delim}.

Computational adequacy now follows easily from the fundamental lemma.

Proof (of Lemma 4). By Lemma 6, we have that $\emptyset \vdash e : \mathbb{N}$ implies that $\emptyset \vDash e : \mathbb{N}$. Now, the statement follows from Theorem 1 and the assumption that $\mathbf{E}[\![e]\!]_{\emptyset}, \sigma_1 \rightsquigarrow^* \mathsf{Ret}\ n, \sigma_2$.

4 Modeling Delimited Continuations

In this section we scale our approach to delimited continuations, which is a challenging example of context-dependent effects. We provide a denotational semantics for a programming language λ_{delim} with `shift`/`reset`, and prove its soundness and adequacy relative to an abstract machine semantics [6]. The semantics and proofs are more complex than for $\lambda_{\mathsf{callcc}}$ due to the nature of delimited continuations and associated type system. To the best of our knowledge, this represents the first formalized sound and adequate direct-style denotational semantics for delimited continuations.

4.1 Syntax and Operational Semantics of λ_{delim}

The syntax and the type system of λ_{delim} is given in Figure 12. It is similar to $\lambda_{\mathsf{callcc}}$, but instead of $\mathsf{call/cc}\ (-.\ -)$ there are operators $\mathcal{D}\ \mathsf{e}$ (delimit the current evaluation context, also known as `reset`) and $\mathcal{S}\ \mathsf{x}.\ e$ (grab the current delimited continuation, and bind it to x in e, also known as `shift`).

$$\begin{array}{rcl}
\langle e\rangle_{\mathsf{term}} & \mapsto & \langle e,\ \square,\ []\rangle_{\mathsf{eval}} \\
\langle K :: mk,\ v\rangle_{\mathsf{mcont}} & \mapsto & \langle K,\ v,\ mk\rangle_{\mathsf{cont}} \\
\langle [],\ v\rangle_{\mathsf{mcont}} & \mapsto & \langle v\rangle_{\mathsf{ret}} \\
\langle \square,\ v,\ mk\rangle_{\mathsf{cont}} & \mapsto & \langle mk,\ v\rangle_{\mathsf{mcont}} \\
\langle K[\square \mathbin{@} v],\ \mathsf{cont}\ K',\ mk\rangle_{\mathsf{cont}} & \mapsto & \langle K',\ v,\ K :: mk\rangle_{\mathsf{cont}} \\
\langle K[e \mathbin{@} \square],\ v,\ mk\rangle_{\mathsf{cont}} & \mapsto & \langle e,\ K[\square \mathbin{@} v],\ mk\rangle_{\mathsf{eval}} \\
\langle v,\ K,\ mk\rangle_{\mathsf{eval}} & \mapsto & \langle K,\ v,\ mk\rangle_{\mathsf{cont}} \\
\langle e_0 \mathbin{@} e_1,\ K,\ mk\rangle_{\mathsf{eval}} & \mapsto & \langle e_1,\ K[e_0 \mathbin{@} \square],\ mk\rangle_{\mathsf{eval}} \\
\langle \mathcal{D}\ \mathsf{e},\ K,\ mk\rangle_{\mathsf{eval}} & \mapsto & \langle e,\ \square,\ K :: mk\rangle_{\mathsf{eval}} \\
\langle \mathcal{S}\ \mathsf{k}.\ e,\ K,\ mk\rangle_{\mathsf{eval}} & \mapsto & \langle e[K/k],\ \square,\ mk\rangle_{\mathsf{eval}}
\end{array}$$

$$\begin{array}{r}
\text{metacontinuations:} \\
\mathit{Mcont} \ni mk ::= [] \mid K :: mk \\
\text{abstract machine config.:} \\
\mathit{Config} ::= \langle e,\ K,\ mk\rangle_{\mathsf{eval}} \mid \\
\langle K,\ v,\ mk\rangle_{\mathsf{cont}} \mid \\
\langle mk,\ v\rangle_{\mathsf{mcont}} \mid \\
\langle e\rangle_{\mathsf{term}} \mid \\
\langle v\rangle_{\mathsf{ret}}
\end{array}$$

Fig. 13. Operational semantics of λ_{delim} (excerpt).

The type system follows Danvy and Filinski [10], 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 $\Gamma \vdash_{\mathsf{pure}} e : \tau$ 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 $\emptyset;\alpha \vdash \mathsf{isprime} : \mathbb{N}/\beta \to \mathbb{B}/\beta;\alpha$. The expression $\mathcal{D}$ $((\mathsf{rec}\,\mathsf{f}(\mathsf{x}) = \mathsf{isprime}\ (\mathcal{S}\ \mathsf{k}.\ \mathsf{x} - 1))\ 2)$ 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 [10] for details.

The operational semantics for λ_{delim} uses a CEK machine, following [6,12,5]. Selected reduction rules appear in Figure 13 (see Coq formalization for the full set of rules). The 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 $\langle e,\ K,\ mk\rangle_{\mathsf{eval}}$, 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

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

Fig. 14. Effects for λ_{delim}.

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

4.2 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 14) 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.

$$\textsc{wp-shift}\quad \frac{\mathsf{has\_state}(\sigma) \qquad \triangleright(\mathsf{has\_state}(\sigma) \twoheadrightarrow \mathsf{wp}\ \beta\ \{\Phi\}) \qquad \blacktriangleright\mathbf{P}(f(\blacktriangleright\kappa)) = \mathsf{next}(\beta)}{\mathsf{wp}\ \kappa(\mathsf{Shift}(f))\ \{\Phi\}}$$

$$\textsc{wp-reset}\quad \frac{\mathsf{has\_state}(\sigma) \qquad \triangleright(\mathsf{has\_state}(\blacktriangleright\kappa :: \sigma) \twoheadrightarrow \mathsf{wp}\ \mathbf{P}(e)\ \{\Phi\})}{\mathsf{wp}\ \kappa(\mathsf{Reset}(\mathsf{next}(e)))\ \{\Phi\}}$$

$$\textsc{wp-pop}\quad \frac{\mathsf{has\_state}(\sigma) \qquad \kappa' = \kappa \text{ if } \sigma = \kappa :: \sigma' \text{ and id otherwise} \qquad \triangleright(\mathsf{has\_state}(\mathtt{tail}(\sigma)) \twoheadrightarrow \mathsf{wp}\ \kappa'(v)\ \{\Phi\})}{\mathsf{wp}\ \mathbf{P}(v)\ \{\Phi\}}$$

$$\textsc{wp-appcont}\quad \frac{\mathsf{has\_state}(\sigma) \qquad \triangleright(\mathsf{has\_state}(\blacktriangleright\kappa :: \sigma) \twoheadrightarrow \mathsf{wp}\ \beta\ \{\Phi\}) \qquad \blacktriangleright\kappa'(e) = \mathsf{next}(\beta)}{\mathsf{wp}\ \kappa(\mathsf{Appcont}(e, \kappa'))\ \{\Phi\}}$$

Fig. 15. Weakest precondition rules for delimited continuations.

$$\mathbf{E}[\![\mathcal{D}\ \mathsf{e}]\!]_\rho = \mathsf{Reset}(\mathbf{P}(\mathbf{E}[\![e]\!]_\rho))$$
$$\mathbf{E}[\![\mathcal{S}\ \mathsf{x}.\ e]\!]_\rho = \mathsf{Shift}(\mathbf{P} \circ (\lambda\kappa.\, \mathbf{E}[\![e]\!]_{\rho, x \mapsto \mathsf{Fun}(\mathsf{next}(\lambda y.\, \mathsf{Tau}(\kappa(\mathsf{next} y))))}))$$
$$\mathbf{E}[\![e_1 \mathbin{@} e_2]\!]_\rho = \mathsf{get\_val}(\mathbf{E}[\![e_2]\!]_\rho, \lambda x.\, \mathsf{get\_fun}(\mathbf{E}[\![e_1]\!]_\rho, \lambda y.\, \mathsf{Appcont}(\mathsf{next}(x), y)))$$
$$\mathbf{V}[\![\mathsf{cont}\ K]\!]_\rho = \mathsf{Fun}(\mathsf{next}(\lambda x.\, \mathsf{Tick}(\mathbf{P}(\mathbf{K}[\![K]\!]_\rho\ x))))$$
$$\mathbf{K}[\![K[\square \mathbin{@} v]]\!]_\rho = \lambda x.\, \mathbf{K}[\![K]\!]_\rho(\mathbf{E}[\![x \mathbin{@} v]\!]_\rho)$$
$$\mathbf{M}[\![mk]\!]_\rho = \mathrm{map}(\lambda k.\, \mathbf{P} \circ \mathbf{K}[\![k]\!]_\rho) mk$$
$$\mathbf{S}[\![\langle e,\ K,\ mk\rangle_{\mathsf{eval}}]\!]_\rho = (\mathbf{P}(\mathbf{E}[\![K[e]]\!]_\rho), \mathbf{M}[\![mk]\!]_\rho)$$
$$\mathbf{S}[\![\langle K,\ v,\ mk\rangle_{\mathsf{cont}}]\!]_\rho = (\mathbf{P}(\mathbf{E}[\![K[v]]\!]_\rho), \mathbf{M}[\![mk]\!]_\rho)$$
$$\mathbf{S}[\![\langle mk,\ v\rangle_{\mathsf{mcont}}]\!]_\rho = (\mathbf{P}(\mathbf{V}[\![v]\!]_\rho), \mathbf{M}[\![mk]\!]_\rho)$$
$$\mathbf{S}[\![\langle e\rangle_{\mathsf{term}}]\!]_\rho = (\mathbf{P}(\mathbf{E}[\![e]\!]_\rho), [])$$
$$\mathbf{S}[\![\langle v\rangle_{\mathsf{ret}}]\!]_\rho = (\mathbf{V}[\![v]\!]_\rho, [])$$

Fig. 16. Denotational semantics for a calculus with delimited control (selected clauses).

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 15. 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 16. 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.

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 2, and put them to use in the soundness theorem:

Theorem 2. *Soundness.* Let $c_0, c_1 \in$ *Config* and suppose $c_0 \mapsto c_1$. Then $\mathbf{S}[\![c_0]\!] \rightsquigarrow^* \mathbf{S}[\![c_1]\!]$.

4.3 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 3. *Adequacy.* Suppose $\emptyset; \mathbb{N} \vdash e : \mathbb{N}; \mathbb{N}$ is a well-typed term, and that $(\mathbf{P}(\mathbf{E}[\![e]\!]_\emptyset), []) \rightsquigarrow^* (\mathsf{Ret}(n), \sigma)$ 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 17. It is again a form of biorthogonal logical relation, with the main focus being the observational refinement $\mathcal{O}$: two configurations are related if they reduce to the same natural number. This coincides with what we want to show in Theorem 3. To facilitate this we then lift $\mathcal{O}$ 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 $\mathcal{O}$-related. Both $\mathcal{M}$ and $\mathcal{O}$ 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 $\mathcal{O}$-related. Finally, we use $\mathcal{K}, \mathcal{M}$, and $\mathcal{O}$ 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 $\mathcal{O}$-related.

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

$$\begin{aligned}
\mathcal{O}\big((\alpha,\kappa,\sigma),(e,K,mk)\big) &\triangleq \begin{array}{l} \mathsf{has\_state}(\sigma) \mathrel{-\!\!*} \\ \mathsf{wp}\,\mathbf{P}(\kappa\ \alpha)\,\big\{\beta.\,\exists v.\,(\langle e,\ K,\ mk\rangle_{\mathsf{eval}} \mapsto^{*} \langle v\rangle_{\mathsf{ret}}) \\ {*}(\beta,v) \in [\![\mathbb{N}]\!] * \mathsf{has\_state}([])\big\} \end{array} \\
\mathcal{M}(P)(\sigma,mk) &\triangleq \forall(\alpha,v).\,P(\alpha,v) \mathrel{-\!\!*} \mathcal{O}((\alpha,\imath,\sigma),(v,\square,mk)) \\
\mathcal{K}(Q,P)(\kappa,K) &\triangleq \begin{array}{l} \square\,\forall(\alpha,v).Q(\alpha,v) \mathrel{-\!\!*} \forall(\sigma,mk).\mathcal{M}(P)(\sigma,mk) \mathrel{-\!\!*} \\ \mathcal{O}((\alpha,\kappa,\sigma),(v,K,mk)) \end{array} \\
\mathcal{E}(P,Q,R)(\beta,e) &\triangleq \begin{array}{l} \forall(\kappa,K).\,\mathcal{K}(P,Q)(\kappa,K) \mathrel{-\!\!*} \forall(\sigma,mk).\,\mathcal{M}(R)(\sigma,mk) \mathrel{-\!\!*} \\ \mathcal{O}((\beta,\kappa,\sigma),(e,K,mk)) \end{array}
\end{aligned}$$

$$\begin{aligned}
[\![\mathbb{N}]\!](\beta,v) &\triangleq \exists n \in \mathbb{N}.\,\beta = \mathsf{Ret}(n) \wedge v = n \\
[\![\tau/\alpha \to \sigma/\beta]\!](\theta,v) &\triangleq \begin{array}{l} \exists F.\,\theta = \mathsf{Fun}(F) \wedge \square\,\forall(\eta,w).\,[\![\tau]\!](\eta,w) \mathrel{-\!\!*} \\ \mathcal{E}([\![\sigma]\!],[\![\alpha]\!],[\![\beta]\!])(\theta \bullet \eta, v\ w) \end{array} \\
[\![\mathit{cont}(\tau,\alpha)]\!](\beta,v) &\triangleq \begin{array}{l} \exists \kappa\ K.\,\beta = \mathsf{Fun}(\mathsf{next}(\lambda x.\,\mathsf{Tick}((\mathbf{P}\circ\kappa)\ x))) \wedge v = \mathsf{cont}\ K \wedge \\ \mathcal{K}([\![\tau]\!],[\![\alpha]\!])(\kappa,K) \end{array} \\
[\![\Gamma]\!](\rho,\gamma) &\triangleq \forall(x:\tau \in \Gamma).\,\square\,\forall\Phi.\,\mathcal{E}([\![\tau]\!],\Phi,\Phi)(\rho(x),\gamma(x)) \\
\Gamma \vDash_{\mathsf{pure}} e : \tau &\triangleq \square\,\forall(\rho,\gamma).\,[\![\Gamma]\!](\rho,\gamma) \mathrel{-\!\!*} \forall\Phi.\,\mathcal{E}([\![\tau]\!],\Phi,\Phi)(\mathbf{E}[\![e]\!]_{\rho}, e[\gamma]) \\
\Gamma;\alpha \vDash e : \tau;\beta &\triangleq \square\,\forall(\rho,\gamma).\,[\![\Gamma]\!](\rho,\gamma) \mathrel{-\!\!*} \mathcal{E}([\![\tau]\!],[\![\alpha]\!],[\![\beta]\!])(\mathbf{E}[\![e]\!]_{\rho}, e[\gamma])
\end{aligned}$$

Fig. 17. Logical relation for λ_{delim}.

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 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 7. *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 $\Gamma \vDash_{\mathsf{pure}} e : \tau$.

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

Fig. 18. Syntax and the new typing rule of the λ_{embed}.

Proof (of Theorem 3). Note that the empty (meta)continuation is related to its denotation: $\mathcal{K}(P, P)(\mathsf{id}, \square)$ and $\mathcal{M}(P)([], [])$ hold for any relation P.

With this, we instantiate $\emptyset; \mathbb{N} \models e : \mathbb{N}; \mathbb{N}$ (that we get from Lemma 7) with the empty continuation/metacontinuation, and get the observational refinement between e and $\mathbf{E}[\![e]\!]$. $\square$

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

5 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.

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 [13] 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 $\mathsf{ref}(\tau)$ and function types $\tau \to \sigma$, with syntax given in Figure 18. 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

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

$$\mathbf{E}[\![\mathsf{embed}\ e]\!]_\rho = \mathsf{Reset}(\mathsf{next}(\mathbf{E}[\![e]\!]_\emptyset)) \qquad \mathbf{E}[\![\mathsf{ref}(e)]\!]_\rho = \mathsf{get\_val}(\mathbf{E}[\![e]\!]_\rho, \lambda x.\mathsf{Alloc}(x, \mathsf{Ret}))$$

$$\mathbf{E}[\![x]\!]_\rho = \rho(x) \qquad \mathbf{E}[\![!\,e]\!]_\rho = \mathsf{get\_val}(\mathbf{E}[\![e]\!]_\rho, \lambda x.\mathsf{Read}(x))$$

$$\mathbf{E}[\![\ell]\!]_\rho = \mathsf{Ret}(\ell) \qquad \mathbf{E}[\![e_1 \leftarrow e_2]\!]_\rho = \mathsf{get\_val}(\mathbf{E}[\![e_2]\!]_\rho, \lambda x. \mathsf{get\_ret}(\mathbf{E}[\![e_1]\!]_\rho, \lambda y.\mathsf{Write}(y, x)))$$

$$\mathrm{VRel} \triangleq \mathbf{IT}^v \to \mathit{iProp} \qquad [\![\tau]\!] : \mathrm{VRel}$$

$$\mathrm{ERel} \triangleq \mathbf{IT} \to \mathit{iProp} \qquad [\![\mathbf{1}]\!](\beta) \triangleq \beta = ()$$

$$\mathcal{O} : \mathrm{VRel} \to \mathrm{ERel} \qquad [\![\mathbb{N}]\!](\beta) \triangleq \exists n.\, \beta = \mathsf{Ret}(n)$$

$$\mathcal{O}(P)(\beta) \triangleq \mathsf{clwp}\ \beta\ \{x.\ P\ x * \mathsf{has\_state}([])\} \qquad [\![\tau \to \sigma]\!](\beta) \triangleq \exists F.\, \beta = \mathsf{Fun}(F) \wedge \Box\, \forall \beta.\, [\![\tau]\!](\beta) \mathrel{-\!\!*} \mathcal{E}([\![\sigma]\!])(\mathsf{Fun}(F) \bullet \beta)$$

$$\mathcal{E} : \mathrm{VRel} \to \mathrm{ERel}$$

$$\mathcal{E}(P)(\beta) \triangleq \mathsf{heap\_ctx} \mathrel{-\!\!*} \mathsf{has\_state}([]) \mathrel{-\!\!*} \mathcal{O}(P)(\beta) \qquad [\![\mathsf{ref}(\tau)]\!](\beta) \triangleq \exists \ell.\, \beta = \mathsf{Ret}(\ell) \wedge \boxed{\exists \nu.\, \ell \mapsto \nu * [\![\tau]\!](\nu)}$$

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

Fig. 19. Denotational semantics (selected clauses) and logical relation for λ_{embed}.

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 $\mathit{State} \triangleq \mathit{State}_{\mathtt{delim}} \times \mathit{State}_{\mathtt{state}}$, and reifiers are defined component-wise. Figure 19 shows the key parts of the denotational semantics. For most of the syntactic constructs we give the standard interpretation. For embed e we use the interpretation $\mathbf{E}[\![-]\!]$ for λ_{delim} from Section 4.2, 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 19), which is similar to the other logical relations we considered in this paper, mainly different in the observation relation $\mathcal{O}$. 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.

Instead of the standard weakest precondition wp, we utilize a *context-local weakest precondition* clwpre which bakes-in the bind rules [28].

Definition 2. *Context-local weakest precondition (clwp) is defined as follows:* $\mathit{clwp}\, \alpha \left\{\Phi\right\} \triangleq \forall \kappa\ (\Psi : \mathbf{IT}^v \to \mathit{iProp}).\, (\forall v.\, \Phi\ v \twoheadrightarrow \mathit{wp}\, (\kappa\ v) \left\{\Psi\right\}) \to \mathit{wp}\, (\kappa\ \alpha) \left\{\Psi\right\}$

Note that clwp always implies wp and validates a form of the bind rule:

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

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 8. *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.

Lemma 9. *Fundamental lemma.* Let $\Gamma \vdash e : \tau$ then $\Gamma \vDash e : \tau$.

Lemma 10. *Denotational type soundness.* Let $\emptyset \vdash e : \tau$ and $(\mathbf{E}[\![e]\!]_{\emptyset}, []) \rightsquigarrow^* (\alpha, \sigma)$, then $(\exists \beta\ \sigma'.\, (\alpha, \sigma) \rightsquigarrow (\beta, \sigma')) \lor (\alpha \in \mathbf{IT}^v)$.

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 20, 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 20.

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].

```
$\texttt{prog} \triangleq \mathsf{Fun}(\mathsf{next}(\lambda y.$
  $\mathsf{Let}\, x = \mathsf{Alloc}(\mathsf{Ret}(1))\, \mathsf{in}$                      Initial offset value
  $\mathsf{Let}\, n = \mathsf{Shift}(\lambda k.\mathsf{next}($                        Capture continuation as $k$
    $\mathsf{Appcont}'(\mathsf{Read}(x), k)$ ;                      First call to $k$ with the init. value of $x$
    $\mathsf{Let}\, m =$                                        $x := x + 1;$
      $\mathsf{NatOp}_{+}(\mathsf{Read}(x), \mathsf{Ret}(1))\, \mathsf{in}\, \mathsf{Write}(x, m)$ ;
    $\mathsf{Appcont}'(\mathsf{Read}(x), k)))$                      Second call to $k$ with updated $x$
    $\mathsf{in}\, \mathsf{get\_ret}(y, (\lambda l.\mathsf{Let}\, p =$
      $\mathsf{NatOp}_{+}(\mathsf{Read}(l), n)\, \mathsf{in}\, \mathsf{Write}(l, p)))))$     $y := y + n;$
```

$$\text{where } \mathsf{Appcont}'(x, y) \triangleq \mathsf{Appcont}(\mathsf{next}(x), \mathsf{next}(\mathsf{Tau} \circ y \circ \mathsf{next}))$$

$$\frac{\mathsf{heap\_ctx} \qquad \mathsf{has\_state}(\sigma) \qquad y \mapsto \mathsf{Ret}(n)}{\mathsf{wp}\ (\mathsf{Reset}(\mathsf{next}(\texttt{prog} \bullet \mathsf{Ret}(y))))\ \{y \mapsto \mathsf{Ret}(n+3) * \mathsf{has\_state}(\sigma)\}}$$

Fig. 20. Example program with delimited continuations and state and its specification.

6 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 $\mathsf{call/cc}$ (x. e) and $\mathcal{S}$ x. e. 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 [34], which would require explicitly first-order representation of functions and continuations, if we want to model first-class continuation. Such 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 are 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 [13].

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 [9] 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 [26] 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 $\texttt{shift}/\texttt{reset}$ is due to Danvy and Filinski [10], where they employ *answer-type modification.* Materzok and Biernacki [22] 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 [17]. Dyvbig, Peyton Jones, and Sabry [11] 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 [5] prove termination for a language with $\mathcal{S}\ \mathsf{x}.\ e$ and $\mathcal{D}\ \mathsf{e}$ (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 $\mathcal{S}\ \mathsf{x}.\ e$ and $\mathcal{D}\ \mathsf{e}$, proving correctness against evaluation-context based operational semantics. In contrast with those works, we define our logical relations on denotations, using the semantics of GITrees and the derived program logics.

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 [29].

We would also like to study other context-dependent effects like exceptions, handlers, and algebraic effects [24,4,33,31,3]. 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 [32].

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.

Data-Availability Statement. The formalisation associated with this paper, which covers all the main results, is available online [27].

References

1. Asai, K.: Logical relations for call-by-value delimited continuations. In: van Eekelen, M.C.J.D. (ed.) 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, pp. 63–78. Intellect (2005)
2. Asai, K., Kameyama, Y.: Polymorphic Delimited Continuations. In: Shao, Z. (ed.) Programming Languages and Systems. pp. 239–254. Springer. https://doi.org/10.1007/978-3-540-76637-7_16
3. Bach Poulsen, C., van der Rest, C.: Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects. Proceedings of the ACM on Programming Languages **7**(POPL), 62:1801–62:1831 (Jan 2023). https://doi.org/10.1145/3571255
4. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. Journal of Logical and Algebraic Methods in Programming **84**(1), 108–123 (Jan 2015). https://doi.org/10.1016/j.jlamp.2014.02.001
5. Biernacka, M., Biernacki, D.: Context-based proofs of termination for typed delimited-control operators. In: Porto, A., López-Fraguas, F.J. (eds.) Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal. pp. 289–300. ACM (2009). https://doi.org/10.1145/1599410.1599446, https://doi.org/10.1145/1599410.1599446
6. Biernacka, M., Biernacki, D., Danvy, O.: 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, https://doi.org/10.2168/LMCS-1(2:5)2005
7. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Abstracting algebraic effects. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290319, https://doi.org/10.1145/3290319
8. Birkedal, L., Møgelberg, R.E., Schwinghammer, J., Støvring, K.: First steps in synthetic guarded domain theory: step-indexing in the topos of trees. Log. Methods Comput. Sci. **8**(4) (2012). https://doi.org/10.2168/LMCS-8(4:1)2012, https://doi.org/10.2168/LMCS-8(4:1)2012
9. Cartwright, R., Felleisen, M.: Extensible denotational language specifications. In: Hagiya, M., Mitchell, J.C. (eds.) Theoretical Aspects of Computer Software. pp. 244–272. Springer, Berlin, Heidelberg (1994). https://doi.org/10.1007/3-540-57887-0_99
10. Danvy, O., Filinski, A.: A functional abstraction of typed contexts. Tech. rep., DIKU – Computer Science Department, University of Copenhagen (1989)
11. Dyvbig, R.K., Peyton Jones, S., Sabry, A.: A monadic framework for delimited continuations **17**(6), 687–730. https://doi.org/10.1017/S0956796807006259, https://www.cambridge.org/core/journals/journal-of-functional-programming/article/monadic-framework-for-delimited-continuations/D99D1394370DFA8EA8428D552B5D8E7E
12. Felleisen, M., Friedman, D.P.: Control operators, the secd-machine, and the λ-calculus. In: Wirsing, M. (ed.) 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. pp. 193–222. North-Holland (1987)
13. Frumin, D., Timany, A., Birkedal, L.: Modular denotational semantics for effects with guarded interaction trees. Proc. ACM Program. Lang. **8**(POPL), 332–361 (2024). https://doi.org/10.1145/3632854, https://doi.org/10.1145/3632854

14. Iris team: The Iris Project website and Coq development (2025), https://iris-project.org/
15. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018). https://doi.org/10.1017/S0956796818000151, https://doi.org/10.1017/S0956796818000151
16. King, A.: Delimited continuation primops. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0313-delimited-continuation-primops.rst (2021), accessed: 2024-06-27
17. Kiselyov, O., Shan, C.c.: A Substructural Type System for Delimited Continuations. In: Della Rocca, S.R. (ed.) Typed Lambda Calculi and Applications. pp. 223–239. Springer, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73228-0_17
18. Koh, N., Li, Y., Li, Y., Xia, L.y., Beringer, L., Honoré, W., Mansky, W., Pierce, B.C., Zdancewic, S.: 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. pp. 234–248. CPP 2019, Association for Computing Machinery, New York, NY, USA (Jan 2019). https://doi.org/10.1145/3293880.3294106
19. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 205–217. ACM (2017). https://doi.org/10.1145/3009837.3009855, https://doi.org/10.1145/3009837.3009855
20. Leijen, D.: Koka: Programming with row polymorphic effect types. In: Levy, P.B., Krishnaswami, N. (eds.) Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014. EPTCS, vol. 153, pp. 100–126 (2014). https://doi.org/10.4204/EPTCS.153.8, https://doi.org/10.4204/EPTCS.153.8
21. Leijen, D.: Structured asynchrony with algebraic effects. In: Lindley, S., Yorgey, B.A. (eds.) Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 3, 2017. pp. 16–29. ACM (2017). https://doi.org/10.1145/3122975.3122977, https://doi.org/10.1145/3122975.3122977
22. Materzok, M., Biernacki, D.: Subtyping delimited continuations **46**(9), 81–93. https://doi.org/10.1145/2034574.2034786, https://doi.org/10.1145/2034574.2034786
23. Pitts, A.M.: Typed operational reasoning. In: Pierce, B.C. (ed.) Advanced Topics In Types And Programming Languages, chap. 7, pp. 245–289. The MIT Press (2004)
24. Plotkin, G.D., Pretnar, M.: Handling Algebraic Effects. Logical Methods in Computer Science **Volume 9, Issue 4** (Dec 2013). https://doi.org/10.2168/LMCS-9(4:23)2013
25. Silver, L., He, P., Cecchetti, E., Hirsch, A.K., Zdancewic, S.: Semantics for Noninterference with Interaction Trees (2023)
26. Sitaram, D., Felleisen, M.: Models of continuations without continuations. In: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 185–196. POPL '91, Association for Computing Machinery. https://doi.org/10.1145/99583.99611, https://dl.acm.org/doi/10.1145/99583.99611
27. Stepanenko, S., Nardino, E., Frumin, D., Timany, A., Birkedal, L.: Context-dependent effects in guarded interaction trees (Jan 2025). https://doi.org/10.5281/zenodo.14623650, https://doi.org/10.5281/zenodo.14623650

28. Timany, A., Birkedal, L.: Mechanized relational verification of concurrent programs with continuations. Proc. ACM Program. Lang. **3**(ICFP), 105:1–105:28 (2019). https://doi.org/10.1145/3341709, https://doi.org/10.1145/3341709
29. Timany, A., Guéneau, A., Birkedal, L.: The logical essence of well-bracketed control flow. Proc. ACM Program. Lang. **8**(POPL), 575–603 (2024). https://doi.org/10.1145/3632862, https://doi.org/10.1145/3632862
30. Timany, A., Krebbers, R., Dreyer, D., Birkedal, L.: A logical approach to type soundness. Journal of the ACM **71** (2024)
31. van den Berg, B., Schrijvers, T., Poulsen, C.B., Wu, N.: Latent Effects for Reusable Language Components. In: Oh, H. (ed.) Programming Languages and Systems. pp. 182–201. Lecture Notes in Computer Science, Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-89051-3_11
32. de Vilhena, P.E., Pottier, F.: A separation logic for effect handlers. Proc. ACM Program. Lang. **5**(POPL) (jan 2021). https://doi.org/10.1145/3434314, https://doi.org/10.1145/3434314
33. Wu, N., Schrijvers, T., Hinze, R.: Effect handlers in scope. In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. pp. 1–12. Haskell '14, Association for Computing Machinery, New York, NY, USA (Sep 2014). https://doi.org/10.1145/2633357.2633358
34. Xia, L., Zakowski, Y., He, P., Hur, C., Malecha, G., Pierce, B.C., Zdancewic, S.: Interaction trees: representing recursive and impure programs in coq. Proc. ACM Program. Lang. **4**(POPL), 51:1–51:32 (2020). https://doi.org/10.1145/3371119, https://doi.org/10.1145/3371119
35. Zakowski, Y., Beck, C., Yoon, I., Zaichuk, I., Zaliva, V., Zdancewic, S.: Modular, compositional, and executable formal semantics for LLVM IR. Proceedings of the ACM on Programming Languages **5**(ICFP), 67:1–67:30 (Aug 2021). https://doi.org/10.1145/3473572
36. Zhang, H., Honoré, W., Koh, N., Li, Y., Li, Y., Xia, L.Y., Beringer, L., Mansky, W., Pierce, B., Zdancewic, S.: Verifying an HTTP Key-Value Server with Interaction Trees and VST. In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving (ITP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 193, pp. 32:1–32:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.32