On Higher-Order Model Checking of Effectful Answer-Type-Polymorphic Programs

TARO SEKIYAMA, National Institute of Informatics, Japan and SOKENDAI, Japan
UGO DAL LAGO, Università di Bologna, Italy and INRIA, France
HIROSHI UNNO, Tohoku University, Japan

Applying higher-order model checking techniques to programs that use effect handlers is a major challenge, given the recent undecidability result obtained by Dal Lago and Ghyselen. This challenge has been addressed by using answer-type modifications, the use of a monomorphic version of which allows to recover decidability. However, the absence of polymorphism leads to a loss of modularity, reusability, and even expressivity. In this work, we study the problem of defining a calculus that on the one hand supports answer-type polymorphism and subtyping but on the other hand ensures the underlying model checking problem to remain decidable. The solution proposed in this paper is based on the introduction of the polymorphic answer-type $\square$ whose role is to provide a good compromise between expressiveness and decidability, the latter demonstrated through the construction of a selective type-directed CPS transformation targeting a calculus *without* effect handlers and any form of polymorphism. Noticeably, the introduced calculus $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$ allows the answer types of effects implemented by tail-resumptive effect handlers to be polymorphic. We also implemented a proof-of-concept model checker for $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$ programs.

CCS Concepts: • **Theory of computation → Type theory**; **Verification by model checking**; • **Software and its engineering → Functional languages**.

Additional Key Words and Phrases: model checking, algebraic effect handlers, answer-type modification

ACM Reference Format:
Taro Sekiyama, Ugo Dal Lago, and Hiroshi Unno. 2025. On Higher-Order Model Checking of Effectful Answer-Type-Polymorphic Programs. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 406 (October 2025), 29 pages. https://doi.org/10.1145/3763184

1 Introduction

Model checking [10] involves systematically exploring all possible states of a system to ensure that it behaves as expected under all conditions. This method is widely used in fields like hardware design, software development, and protocol verification to catch errors early in the design process, before deployment [3, 11]. Since Ong's seminal work [42], model checking of functional programs with higher-order functions and recursion has been known to be decidable when data domains are finite (such as Booleans). The resulting verification technique, called *higher-order model checking* (or HOMC for short), has been used as a starting point for the development of tools capable of verifying functional programs against safety and reachability properties, even in the presence of infinite data domains which can be treated via, e.g., predicate abstraction [9, 27, 51].

Generalizing HOMC to programs involving *computational effects*, such as mutable store, I/O, and exceptions, is crucial to make HOMC more practical and has received attention since the early

Authors' Contact Information: Taro Sekiyama, National Institute of Informatics, Tokyo, Japan and SOKENDAI, Tokyo, Japan, tsekiyama@acm.org; Ugo Dal Lago, Università di Bologna, Bologna, Italy and INRIA, France, ugo.dallago@unibo.it; Hiroshi Unno, Tohoku University, Sendai, Japan, hiroshi.unno@acm.org.

ACM 2475-1421/2025/10-ART406
https://doi.org/10.1145/3763184

times of HOMC [28, 51]. The situation, predictably, is much more complicated than that observed in pure functional programs. In particular, while in the presence of effects such as nondeterministic choice or global state the HOMC problem remains decidable [14, 28], it becomes *undecidable* for effects such as probabilistic choice, exceptions carrying functions, or local stores [31, 32]. Although these extensions with individual effects provide some insights on HOMC for effectful programs, it had been unclear how we can generalize HOMC to deal with a broad class of effects.

Dal Lago and Ghyselen [14] addressed this situation by extending HOMC to *algebraic operations* [46] and *effect handlers* [47, 48].[1] Algebraic operations are effect producers, and the interpretations of the produced effects can be given as, e.g., equational axioms [45], or can be programmed using effect handlers. Depending on how algebraic operations are interpreted, various effects, including nondeterministic and probabilistic choice, global stores, exceptions, and I/O, can be expressed in the framework. When algebraic operations are interpreted via effect handlers, however, the HOMC problem is in general *undecidable* due to the ability of effect handlers to reify *delimited continuations* as program values. On one hand, the use of delimited continuations enables implementing a wide range of effects [4] and leads to the attempts of supporting effect handlers on several programming languages [35, 43, 57] or the development of effect handler libraries [7, 18, 65]. However, on the other hand, their expressivity is *too* powerful to keep HOMC decidable—it enables the encoding of an infinite data domain (such as integers), which makes the HOMC problem undecidable. This issue has prompted the community to try to define conditions on the underlying program allowing both to *capture* interesting classes of programs and to obtain *decidability* results.

This is the path taken by Sekiyama and Unno [56] in their work on model checking functional programs with effect handlers typed via *answer-type modification* (ATM in the following) [12, 15, 24, 38, 55]. Answer types are types of delimited continuations that can be captured by effect handlers, and ATM type systems for effect handlers [12, 24, 55] track how answer types are modified. This tracking makes it possible to reason about *how many* effect handlers are installed in the context enclosing a term. Indeed, the ATM type system of Sekiyama and Unno's calculus[2] $\mathsf{HEPCF}^{\mathsf{ATM}}$ *statically* bounds the number of such effect handlers, and they also show that giving such a static bound is sufficient to guarantee decidability. However, $\mathsf{HEPCF}^{\mathsf{ATM}}$ suffers from a critical limitation: it does not allow any form of *polymorphism*, although some other ATM type systems supports, e.g., *answer-type polymorphism* [24, 38, 55, 61], which improves the reusability and even the expressivity of effectful, well-typed programs. That said, lacking any polymorphism is not surprising, since it is well known that the decidability of HOMC is sensitive to the presence of polymorphism already for *pure* programs [62]. The question then becomes: how far can we go when verifying functional programs with effect handlers *and* some form of polymorphism?

This is precisely the problem this paper addresses. In particular, we introduce a new calculus $\mathsf{HEPCF}^{\mathsf{ATM}}_{\Box}$, which is equipped with effect handlers, an ATM type system, and answer type polymorphism captured by a new type constructor $\Box$. Despite the support for answer-type polymorphism, the HOMC problem remains decidable in $\mathsf{HEPCF}^{\mathsf{ATM}}_{\Box}$, something we proved by defining a type-directed continuation-passing style (CPS) transformation towards the calculus EPCF proposed by Dal Lago and Ghyselen [14], the latter known to admit a decidable HOMC problem. Our CPS transformation is *selective* [41] in that terms with *non*-polymorphic answer types *require* continuations to be evaluated after the transformation, while terms with polymorphic answer types *do not*. In contrast to (non-selective) CPS transformations for effect handlers in the literature [22, 24], which

[1] This work focuses on so-called *dynamically scoped, deep* effect handlers [23]. Other types of effect handlers, such as *shallow* [21, 23] or *lexically scoped* [6, 8] ones, are outside the scope of this work.

[2] The name of (H)EPCF [14] means an extension of PCF [44] with (effect *h*andlers and) algebraic *e*ffects.

Table 1. Feature comparison of the effectful calculi. The column "Effect handlers" indicates whether effect handlers are absent (✗) or present, and, if present, whether they are restricted to be tail-resumptive ($\checkmark^{\mathrm{TR}}$) or not (✓). The column "Polymorphism" indicates whether both answer-type polymorphism and subtyping are supported (✓) or neither is supported (✗).

Calculus	Effect handlers	decidable HOMC	ATM type system?	Polymorphism
EPCF [14]	✗	✓	No	✗
HEPCF [14]	✓	✗	No	✗
GEPCF [14]	$\checkmark^{\mathrm{TR}}$	✓	No	✗
$\mathsf{HEPCF}^{\mathsf{ATM}}$ [56]	✓	✓	Yes	✗
$\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$	✓	✓	Yes	✓

rely on parametric polymorphism in CPS terms, our selective transformation eliminates the need for parametric polymorphism. This distinction is crucial for establishing the decidability of HOMC.

A key technical challenge arising in proving the decidability of the HOMC problem through CPS transformation is that we need to ensure that the CPS transformation preserves the non-termination of source programs, because HOMC can verify not only safety properties but even liveness properties such as termination. To the best of our knowledge, none of the previous works provides a selective CPS transformation for delimited control operators with the guarantee for non-termination preservation (readers interested in the selective CPS transformations in the literature are referred to Section 8). To guarantee non-termination preservation, we show that, if a source term M can reduce, M finally evaluates to some source term N such that the CPS-transformed result of M also finally evaluates to the CPS-transformed result of N *modulo full* $\beta\eta$ *"monadic" reduction* (formulated in Definition 9). Using this property together with the bisimilarity-based reasoning technique proposed by Dal Lago et al. [13], we show that the CPS transformation preserves non-termination.

Answer-type polymorphism does not only solve the monomorphic issue with $\mathsf{HEPCF}^{\mathsf{ATM}}$. It also allows unifying $\mathsf{HEPCF}^{\mathsf{ATM}}$ and GEPCF [14], yet another calculus with effect handlers for which the HOMC problem is decidable. For the sake of guaranteeing decidability, effect handlers in GEPCF are restricted to be *tail-resumptive*—that is, the effect handlers can exclusively call continuations at tail position. While tail-resumptive effect handlers can only define a more restricted class of effects than effect handlers in $\mathsf{HEPCF}^{\mathsf{ATM}}$ (which we call *ATM effect handlers*), they allow a more flexible *use* of effects—terms that only use algebraic operations handled by tail-resumptive effect handlers do not have to limit the number of enclosing effect handlers. We incorporate this flexibility of tail-resumptive effect handlers on the use of effects into our ATM type system by giving answer-type-polymorphic type signatures to algebraic operations handled by tail-resumptive ones. Intuitively, it means that the effect handling by tail-resumptive effect handlers is *purely functional*, so we do not have to be sensitive to the usage of effects handled by them.

We also support another form of polymorphism: *subtyping*. Our subtyping is similar to the form considered by Kawamata et al. [24], but we impose a mild condition to define a selective CPS transformation for the language unifying ATM and tail-resumptive effect handlers.

The comparison of the calculi mentioned above plus HEPCF [14], which supports the full use of effect handlers but instead loses the decidability of the HOMC problem, is summarized in Table 1 (from the perspectives of supported features) and Figure 1 (from the perspective of expressivity). We present examples that can be typechecked in $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$ (extended with subtyping), but not in $\mathsf{HEPCF}^{\mathsf{ATM}}$ nor GEPCF, in Section 2. The expressivity of HEPCF is incomparable with those of $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$ and $\mathsf{HEPCF}^{\mathsf{ATM}}$. On the one hand, HEPCF accommodates programs where the HOMC

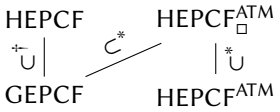


Fig. 1. Expressivity comparison of the calculi with effect handlers. Only comparable calculi are connected. The inclusions $\subset^\dagger$ and $\subset^*$ are proven by Dal Lago and Ghyselen [14] and Theorem 1 in the paper, respectively.

problem is undecidable, while HEPCF$_\square^{\mathsf{ATM}}$ and HEPCF$^{\mathsf{ATM}}$ do not. On the other hand, the ATM type systems of HEPCF$_\square^{\mathsf{ATM}}$ and HEPCF$^{\mathsf{ATM}}$ allow typechecking some terms that are ill typed in non-ATM type systems (see the related work section of [56] for detail). The expressivity of EPCF (not shown in Figure 1) is the same as HEPCF$_\square^{\mathsf{ATM}}$, GEPCF, and HEPCF$^{\mathsf{ATM}}$ because terms in the latter calculi can be CPS-transformed into the former (and the reverse direction holds obviously). However, EPCF is less *macro-expressive* [17] than the other calculi—that is, there is no syntax-directed way to locally translate terms in HEPCF$_\square^{\mathsf{ATM}}$, GEPCF, or HEPCF$^{\mathsf{ATM}}$ to EPCF. Note that HEPCF is strictly more expressive than EPCF due to the undecidability of HOMC in the former.

The contributions of the paper are summarized as follows:

- We introduce HEPCF$_\square^{\mathsf{ATM}}$, which supports ATM and tail-resumptive effect handlers as well as an ATM type system with answer-type polymorphism. HEPCF$_\square^{\mathsf{ATM}}$ enjoys type soundness and is strictly more expressive than HEPCF$^{\mathsf{ATM}}$ and GEPCF.
- We show the decidability of the HOMC problem in HEPCF$_\square^{\mathsf{ATM}}$ via a selective, type-directed CPS transformation that preserves the typing and semantics of HEPCF$_\square^{\mathsf{ATM}}$ programs.
- We extend HEPCF$_\square^{\mathsf{ATM}}$, the CPS transformation, and their metatheory with subtyping.
- The extension with subtyping is implemented as an automated verifier based on the model checker `EffCaml` for HEPCF$^{\mathsf{ATM}}$ [56]. We implement type inference for HEPCF$_\square^{\mathsf{ATM}}$ and the CPS transformation from HEPCF$_\square^{\mathsf{ATM}}$ to EPCF, and integrate them with the higher-order model checker `HorSat2` [9, 30].

The rest of this paper is structured as follows. In Section 2, we briefly explain the motivations leading us to consider the problem of reconciling answer-type polymorphism and decidability of the HOMC problem in more detail. We introduce our calculus HEPCF$_\square^{\mathsf{ATM}}$ in Section 3, define HOMC for HEPCF$_\square^{\mathsf{ATM}}$ in Section 4, and show the decidability of the HOMC problem via the CPS transformation from HEPCF$_\square^{\mathsf{ATM}}$ to EPCF in Section 5. Section 6 briefly explains the extension with subtyping. Section 7 describes our tool for the model checking of HEPCF$_\square^{\mathsf{ATM}}$ terms. We discuss other related works in Section 8 and conclude in Section 9. This paper only states key properties. See the supplementary material for the auxiliary lemmas and detailed proofs, as well as the full definition of the subtyping extension.

Notation. Throughout the paper, we abbreviate a sequence $a_1, \cdots, a_n$ to $\overline{a_i}^{1 \le i \le n}$ or, more simply, $\overline{a}$; its length is denoted by $|\overline{a}|$. Given $\overline{a}$, we write a_i to designate the i-th element of $\overline{a}$.

2 Overview

This section briefly reviews effect handlers and HOMC in the presence of them and presents the benefits of answer-type polymorphism.

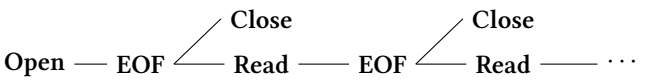
Effect handlers [23, 48] allow programs to be structured so that effect-producing operations, also called *algebraic operations*, are interpreted *from within* the program itself. The key is a construct of the form with H handle M, in which all algebraic operations executed in a term M are interpreted as prescribed by an *effect handler* H. Effect handlers can interpret various effects, such as global store, exceptions, nondeterministic choice, and cooperative multitasking [4], by capturing *delimited*

continuations. For instance, consider the following term given by Bauer and Pretnar [4]:

$$\begin{array}{l}\text{with } H \text{ handle}\\ \quad \text{let } x = \text{if } \mathbf{Decide}\,() \text{ then return } 10 \text{ else return } 20 \text{ in}\\ \quad \text{let } y = \text{if } \mathbf{Decide}\,() \text{ then return } 0 \text{ else return } 5 \text{ in}\\ \quad \text{return } x - y\end{array}$$

which uses the operation **Decide** to make a choice between 10 and 20 and between 0 and 5. The effect handler H identifies how **Decide** is interpreted. To do so, H offers an *operation clause* of the form $\mathbf{Decide}(x; k) \mapsto M$, which takes a parameter x passed in an operation call and the *delimited continuation* k from the call site up to the effect handler H. By invoking the delimited continuation with a value v, the effect handler can resume the computation from the point where **Decide** is called using the value v as a result of the operation call. For example, if H provides a clause $\mathbf{Decide}(x; k) \mapsto k\,\mathsf{true}$, both of the operation calls in the program return true, so the value 10 is returned. Captured delimited continuations can be invoked multiple times. For example, under the effect handler H with clause $\mathbf{Decide}(x; k) \mapsto \text{let } x = k\,\mathsf{true} \text{ in let } y = k\,\mathsf{false} \text{ in return } \max(x, y)$, the program returns 20, the maximum of $x-y$ among all the combinations of $(x, y) \in \{10, 20\}\times\{0, 5\}$.

Although effect handlers are powerful constructs, programs often interact with external environments, e.g., to use storage, transmit some messages, or communicate with the user. One approach to interpreting such algebraic operations, which cannot be interpreted within programs, is to view them as constructors of trees, called *effect trees*, generated by programs. For example, consider operations **Open**, **Close**, **Read**, and **EOF** for file manipulation (for simplicity, we omit file path or descriptor arguments from them), and assume that **EOF** returns a Boolean value indicating if there remains readable data in the file. Given a program $M_{\mathbf{File}} \stackrel{\text{def}}{=} \text{let rec } f\ \_ = \text{if } \mathbf{EOF}\,() \text{ then } \mathbf{Close}\,() \text{ else } (\mathbf{Read}\,(); f\,())\text{ in } (\mathbf{Open}\,(); f\,())$, it generates an effect tree like



This tree represents that the program first calls **Open** followed by **EOF** and branches depending on the result of the call to **EOF**. If it returns true, the program terminates after closing the file, and, otherwise, the program reads the file and repeats the same process from the call to **EOF**.

Through this tree, we can interpret effectful operations by filtering out invalid paths in there. For example, if the file's contents are assumed to be finite, any path involving infinitely many occurrences of **Read** should be invalid, so it is filtered out and then we can conclude that the program *eventually* terminates. Furthermore, it is easy to see that the program uses the file operations correctly—e.g., it calls **EOF**, **Read**, and **Close** after **Open**, calls no file operation after **Close**, and checks if some readable data remains before reading it. In HOMC, the behavior and specifications of algebraic operations not handled by the program can be formulated using *alternating parity tree automata* (APTAs). Interested readers are referred to the supplementary material.

Dal Lago and Ghyselen [14] proposed to conduct this reasoning about effect trees systematically and formally using *higher-order model checking* (HOMC) [29, 42]. In their scheme, effect trees to be verified via HOMC are only constructed by *unhandled* effects, that is, those that have not been handled during the program execution and thus escaped to the top level. However, they show that the full support for effect handlers makes HOMC undecidable, via HEPCF, a variant of PCF that fully supports effect handlers and algebraic effects. They also show that the decidability is gained if effect handlers are restricted to be *tail-resumptive*—i.e., every operation clause has to be in the form $\sigma(x; k) \mapsto \text{let } y = M \text{ in } k\,y$ where k does not occur free in M. To prove this, Dal Lago and Ghyselen introduce GEPCF, a variant of PCF equipped with algebraic effects and tail-resumptive effect

handlers, and define a CPS transformation from GEPCF to EPCF, yet another variant of PCF that is only equipped with algebraic effects, by utilizing the nature of tail-resumptive effect handlers. This CPS transformation enables reducing the HOMC problem for GEPCF to the one for EPCF. Because the HOMC problem for the latter is decidable [14], so is for the former. However, the restriction to tail-resumptive effect handlers is too severe to support the handling of many desirable effects, such as global store, exceptions, and nondeterministic choice. For example, the aforementioned clause $\mathbf{Decide}(x;k) \mapsto \mathsf{let}\, x = k\, \mathsf{true}\, \mathsf{in}\, \cdots$ is not tail-resumptive, thereby rejected by GEPCF.

Alternatively, Sekiyama and Unno [56] identifies the crux of the undecidability as the capability of handling effects by an arbitrary number of *nested* effect handlers, that is, to be simplified, accommodating a program that reaches a term with H_n handle $(\cdots (\mathsf{with}\, H_1\, \mathsf{handle}\, M)\, \cdots)$ during the evaluation for *any* n. Here, the term installs the effect handlers $H_1, \cdots, H_n$ on top of M in a *nested* manner.[3] This capability enables encoding data and operations in an infinite domain via algebraic operations and effect handlers, respectively, hence making the HOMC problem undecidable.

Based on this analysis, Sekiyama and Unno introduced $\mathsf{HEPCF}^{\mathsf{ATM}}$ with a type system that supports *answer types* to allow handling a rich class of effects keeping HOMC decidable. Their type system assigns to a term a *computation type* of the form $\Sigma \triangleright T \,/\, \alpha$, where Σ is a signature of operations that the term may call, T is the type of the values the term may return, and α is an answer type that describes the exact number of effect handlers nested on top of the term.[4] Formally, an answer type is a computation type or a value type (which means that no effect handler can be installed). Namely, in general, its form is described by $\Sigma_1 \triangleright T_1 \,/\, (\cdots (\Sigma_n \triangleright T_n \,/\, T_{n+1}) \cdots)$. Here, for each $i \in [1, n]$, $\Sigma_i \triangleright T_i \,/\, (\cdots)$ is the type of the i-th closest handling construct or, equivalently, it is the return type of the continuation captured by the i-th closest effect handler. Therefore, a term with this answer type requires the nested installation of *exactly n effect handlers*. To see it more concretely, consider a term M with a computation type $\Sigma \triangleright T_0 \,/\, \alpha$ where $\alpha = \Sigma_1 \triangleright T_1 \,/\, (\cdots (\Sigma_n \triangleright T_n \,/\, T_{n+1}) \cdots)$. For each $i \in [1, n]$, the subterm with H_i handle $(\cdots (\mathsf{with}\, H_1\, \mathsf{handle}\, M)\, \cdots)$ can be of the type $\Sigma_i \triangleright T_i \,/\, (\cdots)$—thus, the term with H_n handle $(\cdots (\mathsf{with}\, H_1\, \mathsf{handle}\, M)\, \cdots)$ can be well typed—while with H_{n+1} handle $(\mathsf{with}\, H_n\, \mathsf{handle}\, (\cdots (\mathsf{with}\, H_1\, \mathsf{handle}\, M)\, \cdots))$ is ill typed. Hence, the answer type α exactly bounds the number of effect handlers to be nested on M. This capability of answer types enables us to define a CPS transformation $[\![-]\!]$ from $\mathsf{HEPCF}^{\mathsf{ATM}}$ to EPCF. It transforms an $\mathsf{HEPCF}^{\mathsf{ATM}}$ term of the computation type $\Sigma \triangleright T_0 \,/\, \alpha$ given above to an EPCF term that takes n continuations, each of type $[\![T_{i-1}]\!] \to [\![\Sigma_i \triangleright T_i \,/\, (\cdots)]\!]$ for $i \in [1, n]$. As a result, the HOMC problem for $\mathsf{HEPCF}^{\mathsf{ATM}}$ programs is proven decidable while the desirable effects can be handled.

However, this form of answer types specifies the *exact number* of nested effect handlers, losing reusability and expressivity significantly. For example, consider a function $\lambda x.\mathbf{Decide}\, x$. This function is given a type $\mathsf{unit} \to \Sigma \triangleright \mathsf{bool} \,/\, \alpha$ for some Σ (which specifies the type of **Decide**) and α. As each call to **Decide** is only handled by the closest effect handler, one may expect that a program

$$M_{\mathbf{D}} \stackrel{\text{def}}{=} \mathsf{let}\, f = \mathsf{return}\, \lambda x.\mathbf{Decide}\, x\, \mathsf{in}\, \mathsf{if}\, (\mathsf{with}\, H\, \mathsf{handle}\, \cdots\, f\, ()\, \cdots)\, \mathsf{then}\, f\, ()\, \mathsf{else}\, \cdots$$

can be typechecked, but *it is not* in $\mathsf{HEPCF}^{\mathsf{ATM}}$ since the type of $f\,()$ in the then branch allows installing, e.g., n effect handlers on top of it (n is determined by the answer type α), while the type of the handling construct can only allow installing $n-1$ ones (due to the answer type α assigned to $f\,()$).[5] Thus, the requirement on effect handlers by answer types prevents reusing the same higher-order value in *different* contexts. A more serious problem is that $\mathsf{HEPCF}^{\mathsf{ATM}}$ *may not allow*

[3] The installation nesting of effect handlers can be delayed: it is possible to write a program where at most one effect handler is active during the evaluation, while the result of a handling computation is subsequently handled by another effect handler.
[4] More precisely, computation types in Sekiyama and Unno [56] are in the form $\Sigma \triangleright T \,/\, \alpha_1 \Rightarrow \alpha_2$, which allows modifying the answer type α_1 to α_2, but here we focus on a simpler form where $\alpha = \alpha_1 = \alpha_2$ for explanation.
[5] Furthermore, this example cannot be typechecked in GEPCF either if the effect handler H is not tail-resumptive.

effect handling within recursive functions. For instance, consider a recursive function

$$V_{\mathbf{R}} \stackrel{\text{def}}{=} \mathsf{let\ rec}\ f\ \_ = \mathsf{with}\ H\ \mathsf{handle}\ \mathsf{if}\ \mathbf{EOF}\ ()\ \mathsf{then}\ \mathbf{Raise}\ ()\ \mathsf{else}\ (\mathbf{Read}\ (); f\ ())\ ,$$

which raises an exception via the operation **Raise** handled by the effect handler H if no readable data remains. This recursive function cannot be typechecked in $\mathsf{HEPCF}^{\mathsf{ATM}}$ because the call to the function f will install an arbitrary number of effect handlers atop some term, despite the fact that the HOMC problem is decidable for programs only with handleable exceptions carrying first-order values and any other unhandled effects [51].

We address this situation by extending $\mathsf{HEPCF}^{\mathsf{ATM}}$ with *answer-type polymorphism* and subtyping [24, 38, 55], which enable us to abstract the typing information about effect handlers. Our answer-type polymorphism follows the formalism of Kawamata et al. [24]. In the presence of it, an answer type α is either a computation type or a new type constructor $\square$, which can polymorphically be replaced by any answer type—even one involving $\square$ itself. For example, the type $\square$ can be coerced into $\Sigma_1 \triangleright T_1 \,/\, \square$, which in turn can be coerced to $\Sigma_1 \triangleright T_1 \,/\, (\Sigma_2 \triangleright T_2 \,/\, \square)$, and so on. This capability of $\square$ *allows the context to install zero or more effect handlers to be nested*. For instance, the function $\lambda x.\mathbf{Decide}\, x$ can be given type $\mathsf{unit} \to \Sigma_1 \triangleright \mathsf{bool} \,/\, (\Sigma_2 \triangleright \mathsf{bool} \,/\, \square)$ for some Σ_1, which determines the type signature of **Decide**, and Σ_2. The answer type $\Sigma_2 \triangleright \mathsf{bool} \,/\, \square$ of the return type requires the context to install *at least* one effect handler and *allows* it to nest two or more ones atop the application of the function. Using this type, the program $M_{\mathbf{D}}$ *can* be typechecked because $f\,()$ can allow its context to install two or more effect handlers and, therefore, the handling construct $\mathsf{with}\ H\ \mathsf{handle}\ \cdots f\,()\ \cdots$ can allow the enclosing, outer context to install at least one effect handler, which is aligned with the requirement of $f\,()$ in the then branch. Similarly, answer-type polymorphism enables the function $V_{\mathbf{R}}$ to be typechecked, as follows. First, the call to **Raise** can be of type $\Sigma_1 \triangleright \mathsf{unit} \,/\, (\Sigma_2 \triangleright \mathsf{unit} \,/\, \square)$ for some Σ_1, which determines the type signature of **Raise**, and Σ_2. This type means that the call to **Raise** requires installing at least one effect handler for exception handling. Then, since the handling construct $\mathsf{with}\ H\ \mathsf{handle}\ \mathsf{if}\ \mathbf{EOF}\ ()\ \mathsf{then}\ \mathbf{Raise}\ ()\ \mathsf{else}\ (\mathbf{Read}\ (); f\ ())$ installs the effect handler H on top of $f\,()$, it can have the type $\Sigma_2 \triangleright \mathsf{unit} \,/\, \square$, which indicates that the outer context can nest the installation of zero or more effect handlers. Therefore, the recursive function may install an arbitrary number of effect handlers on top of the handling construct.[6]

Interestingly, answer-type polymorphism can also accommodate the flexibility of tail-resumptive effect handlers. When an algebraic operation σ is handled by a tail-resumptive clause with the answer type $\square$, we can give the operation σ a signature of the form $T_1 \rightsquigarrow T_2 \,/\, \square$, which means that the operation takes an argument of type T_1 and returns a value of type T_2, and the answer type of the operation call is polymorphic. This is valid because, in essence, tail-resumptive operation clauses are simply functions. Therefore, if they imposes no requirements on enclosing effect handlers—i.e., their answer types are $\square$—calls to them do not either. Such type assignment is useful in dealing with algebraic operations that we do not expect to be handled (thus, do expect to escape and construct effect trees) but are called in the scope of effect handlers. For example, recall that the interpretations of **EOF** and **Read** do not expect to be programmed by effect handlers. Thus, what the effect handler H in the function $V_{\mathbf{R}}$ can do is only to *forward* them to the outer effect handler, as $\mathbf{EOF}(x; k) \mapsto \mathsf{let}\ y = \mathbf{EOF}\ x\ \mathsf{in}\ k\ y$. This clause is tail-resumptive and its answer type can be $\square$ as **EOF** should only be forwarded to the top level. Therefore, the signature of **EOF** under the effect handler H can be $\mathsf{unit} \rightsquigarrow \mathsf{bool} \,/\, \square$, which indicates that **EOF** can be called freely in any context.

The answer type $\square$ imposes no requirement or constraint on the context—particularly, regarding the number of nested effect handlers—except that any effect invoked by the term can be handled only by a tail-resumptive clause with the answer type $\square$. When the invoked effects are to be handled

[6]Furthermore, we also require Σ_2 can be coerced into Σ_1 via subtyping to typecheck $V_{\mathbf{R}}$. See Example 6.2 for details.

$$
\begin{array}{rrcl}
\textbf{Variables} & x, y, z, f, h, k & \textbf{Algebraic operations} & \sigma, \varsigma \\
\textbf{Base constants} & c & ::= & \mathsf{true} \mid \mathsf{false} \mid () \mid \cdots \\
\textbf{Enum constants} & \varepsilon & ::= & \underline{1} \mid \underline{2} \mid \cdots \\
\textbf{Values} & V, W & ::= & x \mid c \mid \varepsilon \mid \lambda x.M \mid \mathsf{fix}\, x.V \\
\textbf{Terms} & L, M, N & ::= & \mathsf{return}\, V \mid \mathsf{let}\, x = M\, \mathsf{in}\, N \mid V\ W \mid \mathsf{case}(V; M_1, \cdots, M_n) \mid \\
 & & & \sigma(V; x.\, M) \mid \mathsf{with}\, H\, \mathsf{handle}\, M \\
\textbf{Handlers} & H & ::= & \{\mathsf{return}\, x \mapsto L\} \uplus \{\sigma_i(x_i; k_i) \mapsto M_i\}^{1 \le i \le m} \uplus \{\varsigma_i(y_i) \mapsto N_i\}^{1 \le i \le n}
\end{array}
$$

Fig. 2. Program Syntax.

by other—especially, non-tail-resumptive—effect handlers, the answer type must be a computation type, determining an upper bound on the number of nested effect handlers that handle the effects in a "non-functional" manner. This enables us to define a novel CPS transformation from programs well-typed in our type system to EPCF, thereby ensuring the decidability of their model checking.

3 $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$: Finitary PCF with Effect Handlers and Answer-Type Polymorphism

This section introduces $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$, a finitary[7] variant of fine-grain call-by-value PCF [36, 44] with effect handlers, an ATM type system, and answer-type polymorphism. We first define its syntax, operational semantics, and type system and then show its basic properties. Typing examples are given in Section 6. We highlight in gray boxes the parts extended or modified from $\mathsf{HEPCF}^{\mathsf{ATM}}$ [56].

3.1 Syntax

The program syntax of $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$, presented in Figure 2, is the same as that of HEPCF [14] except for the presence of tail-resumptive effect handlers. Programs are classified as either values or terms.

Values, ranged over by V and W, are canonical forms not being evaluated further, including variables x, base constants c, enum constants ε, functions $\lambda x.M$, and the fixed-point operator $\mathsf{fix}\, x.V$. Base constants are inhabitants in some finite data domains. For example, Boolean values and the unit value can be base constants. Enum constants are natural numbers, used to implement case analysis. As we will see shortly, the type of enum constants specifies an upper bound to the number of its inhabitants. Thus, well-typed $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ programs can only access finite data domains.

Terms, ranged over by L, M, and N, may perform possibly effectful computations. Most constructs are standard, e.g., a return-value construct $\mathsf{return}\, V$ embeds the value V into a term, and a case construct $\mathsf{case}(V; M_1, \cdots, M_n)$ does case analysis on the enum value V. An *algebraic operation call*, or simply operation call, $\sigma(V; x.\, M)$ involves the parameter value V and the continuation $x.\, M$, where the continuation takes the result of the operation call as its argument x (thus, x is bound in M). The continuation will be reified and passed on to an effect handler that handles the operation call (if any). A handling construct $\mathsf{with}\, H\, \mathsf{handle}\, M$ handles calls to algebraic operations in the term M using the effect handler H; we call M the *handled term*. An effect handler consists of one return clause $\mathsf{return}\, x \mapsto L$, zero or more (non-tail-resumptive) operation clauses $\{\sigma_i(y_i; k_i) \mapsto M_i\}^{1 \le i \le m}$, which we call *answer-type-modifying* (ATM), and zero or more tail-resumptive operation clauses $\{\varsigma_i(z_i) \mapsto N_i\}^{1 \le i \le n}$. The body L of the return clause is evaluated when the handled term M is evaluated to a value, to which L refers by the variable x. The ATM operation clause $\sigma_i(y_i; k_i) \mapsto M_i$ is executed when the handled term M calls the algebraic operation σ_i. The clause takes the parameter of the operation call as y_i and the reified delimited continuation as k_i. The tail-resumptive operation clause $\varsigma_i(z_i) \mapsto N_i$ only takes the parameter z_i. Semantically, it is the same as the operation clause

[7] All available data domains are finite.

Evaluation rules $\boxed{M_1 \longrightarrow M_2}$

$$
\begin{array}{rcll}
(\lambda x.M)\ V & \longrightarrow & M[V/x] & \\
(\mathsf{fix}\, x.V)\ W & \longrightarrow & V[\mathsf{fix}\, x.V/x]\ W & \\
\mathsf{case}(\underline{\mathrm{i}}; M_1, \cdots, M_n) & \longrightarrow & M_i & (\text{if } 0 < i \leq n) \\
\mathsf{let}\ x = \mathsf{return}\ V_1\ \mathsf{in}\ M_2 & \longrightarrow & M_2[V_1/x] & \\
\mathsf{let}\ x = \sigma(V_1; y.\, M_1)\ \mathsf{in}\ M_2 & \longrightarrow & \sigma(V_1; y.\, \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2) & (\text{if } y \notin fv(M_2)) \\
\mathsf{with}\ H\ \mathsf{handle}\ \mathsf{return}\ V & \longrightarrow & M[V/x] & (\text{if } \mathsf{return}\, x \mapsto M \in H) \\
\mathsf{with}\ H\ \mathsf{handle}\ \sigma(V; y.\, M) & \longrightarrow & N[V/x][\lambda y.\mathsf{with}\ H\ \mathsf{handle}\ M/k] & (\text{if } \sigma(x;k) \mapsto N \in H) \\
\mathsf{with}\ H\ \mathsf{handle}\ \sigma(V; y.\, M) & \longrightarrow & \mathsf{let}\ y = N[V/x]\ \mathsf{in}\ \mathsf{with}\ H\ \mathsf{handle}\ M & (\text{if } \sigma(x) \mapsto N \in H)
\end{array}
$$

$$
\frac{M \longrightarrow N}{\mathsf{let}\ x = M\ \mathsf{in}\ L \longrightarrow \mathsf{let}\ x = N\ \mathsf{in}\ L}
\qquad
\frac{M \longrightarrow N}{\mathsf{with}\ H\ \mathsf{handle}\ M \longrightarrow \mathsf{with}\ H\ \mathsf{handle}\ N}
$$

Fig. 3. Semantics.

$\varsigma_i(z_i; k_i) \mapsto \mathsf{let}\ x = N_i\ \mathsf{in}\ k_i\, x$ where k_i does not occur free in N_i—thus, it enforces the continuation to be called at tail position. The syntactic distinction between ATM and tail-resumptive operation clauses allows assigning answer-type-polymorphic type signatures only to the latter.

The notions of free variables and capture-avoiding value substitution are defined as usual. The metafunction fv returns a set of free variables occurring in a given term or value. We also write $M[W/x]$ and $V[W/x]$ for the term and value obtained by substituting the value W for the free variable x in the term M and the value V, respectively.

Note that the program examples described in Section 2 are easily rewritten into $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$. For instance, an operation call $\sigma\ V$ there is written as $\sigma(V; x.\, \mathsf{return}\, x)$ in $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$, and recursive functions can be expressed using fix and functions.

3.2 Operational Semantics

The call-by-value operational semantics of $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ is given as the evaluation relation $\longrightarrow$, which is the smallest binary relation over terms that is closed under the rules in Figure 3. Except for the tail-resumptive handling of algebraic operations, all the evaluation rules are the same as those from HEPCF. A call to an algebraic operation σ moves up towards the closest handling construct. If the algebraic operation is handled by an ATM operation clause $\sigma(x; k) \mapsto M$, the continuation involved in the operation call is reified and then the clause's body M is evaluated after substituting the parameter value for x and the reified continuation for k. Note that the underlying handler is responsible for the handling of the operations called by the continuation. If the algebraic operation is handled by a tail-resumptive operation clause $\sigma(x) \mapsto M$, the continuation is *not* reified and only the value resulting from the evaluation of the body M is passed to the continuation.

3.3 Type System

3.3.1 Types. The syntax of types for $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ is shown in Figure 4, consisting of value and computation types. Value types, ranged over by T and U, are for values. Base and enum types are the types of base and enum constants, respectively. Function types $T \to C$ are assigned to functions. The number n in an enum type n is simply the number of its inhabitants. Computation types, ranged over by C, are assigned to terms. In a computation type $\Sigma \triangleright T \,/\, A$ assigned to a term, the *operation signature* Σ specifies algebraic operations the term may call, the value type T specifies the value returned by the term, and the *control effect* A specifies the context enclosing the term up

$$
\begin{array}{rrcl}
\textbf{Base types} & B & ::= & \mathsf{bool} \mid \mathsf{unit} \mid \cdots \\
\textbf{Enum types} & E & ::= & \underline{1} \mid \underline{2} \mid \cdots \\
\textbf{Value types} & T, U & ::= & B \mid E \mid T \to C \\
\textbf{Computation types} & C, D & ::= & \Sigma \triangleright T \,/\, A \\
\textbf{Control effects} & A & ::= & \boxed{\square} \mid C_1 \Rightarrow C_2 \\
\textbf{Operation signatures} & \Sigma & ::= & \{\sigma_i : T_i^{\mathrm{par}} \rightsquigarrow T_i^{\mathrm{ari}} \,/\, A_i\}^{1 \le i \le n} \\
\textbf{Typing contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x : T
\end{array}
$$

Fig. 4. Type Syntax.

to the delimiter—that is, the closest handling construct.[8] For $C = \Sigma \triangleright T \,/\, A$, we designate Σ, T, and A by $C.\Sigma$, $C.T$, and $C.A$, respectively.

A control effect A is either the *answer-type-polymorphic* (ATP) effect $\square$ or an *answer-type-modifying* (ATM) effect $C^{\mathrm{ini}} \Rightarrow C^{\mathrm{fin}}$ (where C^{ini} and C^{fin} are computation types called the *initial* and *final answer types* respectively), which allows the modification of answer types from the initial to the final one—this is called *answer-type modification*.[9] The ATP effect $\square$ means that the effects invoked by a term, if any, must be handled in a functional manner, that is, by tail-resumptive operation clauses. Thus, an operation call $\sigma(V; x.\, M)$ has the ATP effect $\square$ only if there is no enclosing effect handler or the closest enclosing effect handler provides a tail-resumptive clause for σ. The type checking propagates the latter information via the operation signature, as seen shortly. Note that "pure" terms, like return-value constructs, can also have the ATP effect $\square$ because they invoke no effect. By contrast, an ATM effect $C_1 \Rightarrow C_2$ is assigned to a term whose effects can be handled by ATM effect handlers. It is a generalization of answer types explained in Section 2; an answer type C with a computation type C in there is expressed as $C \Rightarrow C$ in $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$. The meaning of $C_1 \Rightarrow C_2$ is twofold. First, when the term calls an algebraic operation that captures the continuation, the continuation has to behave as specified by the type C_1. Second, in the course of evaluating the term, the enclosing context (up to the nearest delimiter) is transformed together with the term into another term that behaves as specified by the type C_2. For example, consider a handling term

$$\mathsf{with}\, H\, \mathsf{handle}\, \sigma((); z.\, \mathsf{return}\, z)$$

where $H = \{\mathsf{return}\, x \mapsto \mathsf{return}\, x\} \uplus \{\sigma(x; k) \mapsto \mathsf{let}\, y = k\, \underline{1}\, \mathsf{in}\, \mathsf{case}(y; \mathsf{return}\, \mathsf{true}, \mathsf{return}\, \mathsf{false})\}$. The evaluation of the handling term starts by calling the algebraic operation σ in the handled term, and the operation clause of σ provided by the effect handler H takes the reified continuation $\lambda z.\mathsf{with}\, H\, \mathsf{handle}\, \mathsf{return}\, z$ as the variable k. Because the operation call is handled by the ATM effect handler H, it must be given an ATM effect $C^{\mathrm{ini}} \Rightarrow C^{\mathrm{fin}}$ for some C^{ini} and C^{fin}. The initial answer type C^{ini} specifies the behavior of the reified continuation $\lambda z.\mathsf{with}\, H\, \mathsf{handle}\, \mathsf{return}\, z$. Since the continuation takes and returns the enum constant $\underline{1}$, and invokes no effect, its type can be, e.g., a function type $2 \to \Sigma \triangleright 2 \,/\, \square$ with some operation signature Σ, where the return type $\Sigma \triangleright 2 \,/\, \square$ corresponds to C^{ini}. Furthermore, the operation call is rewritten to the body of the operation clause. Therefore, the final answer type C^{fin} matches with the type of the body term. In the example, the body can be given the type $\Sigma \triangleright \mathsf{bool} \,/\, \square$ as it returns a Boolean value and invokes no effect. As a result, the control effect assigned to the operation call can be $(\Sigma \triangleright 2 \,/\, \square) \Rightarrow (\Sigma \triangleright \mathsf{bool} \,/\, \square)$. As

[8] There is no significant difference between contexts and continuations, but we use continuations to refer to a functional representation of contexts.

[9] In the previous work [24, 56], the ATP effect is called *pure* and ATM effects are called *impure*. We adopt the new names for them because even terms with $\square$ may call algebraic operations and we think that the names ATP and ATM better capture the intuition behind the control effects.

shown via the example, ATM control effects can precisely track the transition of control flow in the presence of effect handlers capable of flexible context manipulation.

An operation signature specifies the interface of algebraic operations, associating each algebraic operation σ that may be invoked, with a type of the form $T^{\mathrm{par}} \rightsquigarrow T^{\mathrm{ari}} \,/\, A$. The value types T^{par} and T^{ari} represent the input and output of the operation σ. Because the input is a parameter of the algebraic operation and the output corresponds to the arity of the captured delimited continuations, T^{par} and T^{ari} are called *parameter* and *arity types*, respectively. The control effect A describes what operation clause handles a call to σ. If $A = \Box$, σ is called *answer-type-polymorphic (ATP)* and its invocation should be handled by a tail-resumptive clause; otherwise, σ is called *answer-type-modifying* (ATM) and its invocation should be handled by an ATM clause with the same control effect as A. Therefore, for the above example term, the operation signature of the handled term can be $\{\sigma : \mathsf{unit} \rightsquigarrow 2 \,/\, (\Sigma \triangleright 2 \,/\, \Box) \Rightarrow (\Sigma \triangleright \mathsf{bool} \,/\, \Box)\}$ since σ takes the unit value, returns the enum constant inhabited by the type 2, and its control behavior is specified by the control effect $(\Sigma \triangleright 2 \,/\, \Box) \Rightarrow (\Sigma \triangleright \mathsf{bool} \,/\, \Box)$ as explained above. In contrast, e.g., for an operation **EOF**, which should be handled by a tail-resumptive clause $\mathbf{EOF}(x) \mapsto \mathbf{EOF}\,(x)$ to be forwarded to the top level, a type signature $\mathsf{unit} \rightsquigarrow \mathsf{bool} \,/\, \Box$ can be given.

3.3.2 Typing Rules. The type system consists of typing judgments for values $\Gamma \vdash V : T$ and for terms $\Gamma \vdash M : C$. The typing rules are presented in Figure 5. Typing contexts Γ are sequences of bindings of the form $x : T$ that assigns the type T to the variable x. We assume that the same variable is bound only once in the same typing context. The typing rules for values are self-explanatory and it is easy to see how they capture the aforementioned ideas for values. The metafunction ty assigns a base type to each base constant. The typing rules for terms are similar to those given by Kawamata et al. [24] except that subtyping is absent (although added in Section 6) and the typing of operation calls and handling takes the presence of $\Box$ in operation signatures into account.

As mentioned above, the control effects of value-return constructs can be ATP. One can embed terms with the ATP effect $\Box$ into the context requiring an ATM effect (HT_Emb). Since ATP terms do not change the context, the final answer type is the same as the initial one. The rules (HT_App) and (HT_Case) for function applications and case constructs naturally reflects their behavior.

For a let construct $\mathsf{let}\, x = M_1 \,\mathsf{in}\, M_2$, if the term M_1 has the ATP effect (i.e., it does not change the enclosing context), how the let construct changes the context is determined by the term M_2 (HT_Let). If both of the control effects of M_1 and M_2 are ATM, say $C_1^{\mathrm{ini}} \Rightarrow C_1^{\mathrm{fin}}$ and $C_2^{\mathrm{ini}} \Rightarrow C_2^{\mathrm{fin}}$, then so is that of the let construct (HT_LetATM). The form of an ATM effect of the let construct is determined as follows. First, the evaluation of the let construct starts by evaluating M_1 and the final answer type C_1^{fin} of M_1 describes how the context enclosing M_1 changes. Thus, the change of the context enclosing the let construct is determined by C_1^{fin}, i.e., the final answer type of the let construct is C_1^{fin}. The initial answer type C_1^{ini} is the requirement for the context enclosing M_1. Because M_1 is placed under the context $\mathsf{let}\, x = [\,]\, \mathsf{in}\, M_2$ (where $[\,]$ is the hole of the context) and its control behavior is specified by the final answer type C_2^{fin}, the requirement C_1^{ini} has to be implied by the guarantee C_2^{fin}. The rule (HT_LetATM) enforces this demand by imposing $C_1^{\mathrm{ini}} = C_2^{\mathrm{fin}}$. Finally, the requirement for the outer context is specified by the initial answer type C_2^{ini} of the term M_2 since M_2 may perform operation calls that reify delimited continuations involving the outer context. Thus, the initial answer type of the let construct corresponds to C_2^{ini}.

The typing of an operation call $\sigma(V; x.\, M)$ depends on the type signature of the algebraic operation σ. If σ is ATP, then the control effect of the operation call is also ATP. Because the continuation $x.\, M$ stands for the context of the operation call, the type-checking of $\sigma(V; x.\, M)$ is carried out in a way similar to (HT_Let). Otherwise, the operation call is given an ATM effect, and then the type-checking is similar to the one done by (HT_LetATM). Note that: the parameter

Typing rules for values $\boxed{\Gamma \vdash V : T}$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ \text{HT\_Var} \qquad \frac{}{\Gamma \vdash c : ty(c)}\ \text{HT\_Const} \qquad \frac{0 < i \leq n}{\Gamma \vdash \underline{i} : \mathsf{n}}\ \text{HT\_EConst}$$

$$\frac{\Gamma, x : T \vdash M : C}{\Gamma \vdash \lambda x.M : T \to C}\ \text{HT\_Abs} \qquad \frac{\Gamma, x : T \to C \vdash V : T \to C}{\Gamma \vdash \mathsf{fix}\, x.V : T \to C}\ \text{HT\_Fix}$$

Typing rules for terms $\boxed{\Gamma \vdash M : C}$

$$\frac{\Gamma \vdash V : T}{\Gamma \vdash \mathsf{return}\, V : \Sigma \triangleright T \,/\, \square}\ \text{HT\_Return} \qquad \frac{\Gamma \vdash M : \Sigma \triangleright T \,/\, \square}{\Gamma \vdash M : \Sigma \triangleright T \,/\, C \Rightarrow C}\ \text{HT\_Emb}$$

$$\frac{\Gamma \vdash M_1 : \Sigma \triangleright T_1 \,/\, \square \quad \Gamma, x : T_1 \vdash M_2 : \Sigma \triangleright T_2 \,/\, A}{\Gamma \vdash \mathsf{let}\, x = M_1 \,\mathsf{in}\, M_2 : \Sigma \triangleright T_2 \,/\, A}\ \text{HT\_Let}$$

$$\frac{\Gamma \vdash M_1 : \Sigma \triangleright T_1 \,/\, C \Rightarrow C^{\mathrm{fin}} \quad \Gamma, x : T_1 \vdash M_2 : \Sigma \triangleright T_2 \,/\, C^{\mathrm{ini}} \Rightarrow C}{\Gamma \vdash \mathsf{let}\, x = M_1 \,\mathsf{in}\, M_2 : \Sigma \triangleright T_2 \,/\, C^{\mathrm{ini}} \Rightarrow C^{\mathrm{fin}}}\ \text{HT\_LetATM}$$

$$\frac{\Gamma \vdash V_1 : T \to C \quad \Gamma \vdash V_2 : T}{\Gamma \vdash V_1\ V_2 : C}\ \text{HT\_App} \qquad \frac{\Gamma \vdash V : \mathsf{n} \quad \forall i \in [1, n].\ \Gamma \vdash M_i : C}{\Gamma \vdash \mathsf{case}(V; M_1, \cdots, M_n) : C}\ \text{HT\_Case}$$

$$\frac{\Sigma \ni \sigma : T^{\mathrm{par}} \rightsquigarrow T^{\mathrm{ari}} \,/\, \square \quad \Gamma \vdash V : T^{\mathrm{par}} \quad \Gamma, x : T^{\mathrm{ari}} \vdash M : \Sigma \triangleright T \,/\, A}{\Gamma \vdash \sigma(V; x.M) : \Sigma \triangleright T \,/\, A}\ \text{HT\_Op}$$

$$\frac{\Sigma \ni \sigma : T^{\mathrm{par}} \rightsquigarrow T^{\mathrm{ari}} \,/\, C^{\mathrm{ini}} \Rightarrow C^{\mathrm{fin}} \quad \Gamma \vdash V : T^{\mathrm{par}} \quad \Gamma, x : T^{\mathrm{ari}} \vdash M : \Sigma \triangleright T \,/\, C \Rightarrow C^{\mathrm{ini}}}{\Gamma \vdash \sigma(V; x.M) : \Sigma \triangleright T \,/\, C \Rightarrow C^{\mathrm{fin}}}\ \text{HT\_OpATM}$$

$$\frac{\begin{array}{c} H = \{\mathsf{return}\, x \mapsto L\} \uplus \{\sigma_i(y_i; k_i) \mapsto M_i\}^{1 \leq i \leq m} \uplus \{\varsigma_i(z_i) \mapsto N_i\}^{1 \leq i \leq n} \\ \Sigma = \{\sigma_i : T_i^{\mathrm{par}} \rightsquigarrow T_i^{\mathrm{ari}} \,/\, C_i^{\mathrm{ini}} \Rightarrow C_i^{\mathrm{fin}}\}^{1 \leq i \leq m} \uplus \{\varsigma_i : U_i^{\mathrm{par}} \rightsquigarrow U_i^{\mathrm{ari}} \,/\, \square\}^{1 \leq i \leq n} \\ \Gamma \vdash M : \Sigma \triangleright T \,/\, C^{\mathrm{ini}} \Rightarrow C^{\mathrm{fin}} \quad \Gamma, x : T \vdash L : C^{\mathrm{ini}} \\ \forall i \in [1, m].\ \Gamma, y_i : T_i^{\mathrm{par}}, k_i : T_i^{\mathrm{ari}} \to C_i^{\mathrm{ini}} \vdash M_i : C_i^{\mathrm{fin}} \\ \forall i \in [1, n].\ \Gamma, z_i : U_i^{\mathrm{par}} \vdash N_i : \Sigma_i \triangleright U_i^{\mathrm{ari}} \,/\, \square \\ \forall C \in \{\overline{C_i^{\mathrm{ini}}}^{1 \leq i \leq m}, C^{\mathrm{fin}}\}.\ \forall i \in [1, n].\ C.\Sigma = \Sigma_i \end{array}}{\Gamma \vdash \mathsf{with}\, H \,\mathsf{handle}\, M : C^{\mathrm{fin}}}\ \text{HT\_Handle}$$

Fig. 5. Type System.

value V has to be of the parameter type T^{par}; the bound variable x of the continuation is given the arity type T^{ari} since the continuation may refer to the result of the operation call via x; and the return value of the operation call is determined by the continuation.

The typing rule (HT_Handle) for handling constructs is definitely the most complicated. Consider a handling construct with H handle M to be typechecked. First, (HT_Handle) assumes that the handled term M has a computation type $\Sigma \triangleright T \,/\, C^{\mathrm{ini}} \Rightarrow C^{\mathrm{fin}}$. Then, the first two premises of the rule require the effect handler H to implement all the algebraic operations in Σ that may be called by the handled term M. It also requires the return clause's body L has the type C^{ini} because the return clause is the context for the handled term M. By contrast, because the final answer type C^{fin} specifies terms to which the context of M changes, it is assigned to the handling construct. The operation clauses are also typechecked to ensure that their bodies behave as specified by the

operation signature Σ. Note that the control effect of the clause of an ATP operation ς_i has to be ATP to ensure that the operations invoked by the clause do not influence the outer context. Finally, for the clause $\varsigma_i(z_i) \mapsto M_i$ of each ς_i, its operation signature Σ_i is required to be equal to $C^{\mathrm{fin}}.\Sigma$ and $C_j^{\mathrm{ini}}.\Sigma$ (here, C_j^{ini} is the initial answer type of the ATM operation σ_j) for subject reduction. The equality $\Sigma_i = C^{\mathrm{fin}}.\Sigma$ is imposed to make the handling of ς_i type-preserving. When the handled term M calls ς_i along with parameter value V and continuation $y.\, N$, the handing construct is evaluated to let $y = M_i[V/z_i]$ in with H handle N. Here, the term with H handle N has the operation signature $C^{\mathrm{fin}}.\Sigma$, so the operation signature of the term $M_i[V/z_i]$ also has to be $C^{\mathrm{fin}}.\Sigma$, according to (HT_Let). This can be enforced by requiring $\Sigma_i = C^{\mathrm{fin}}.\Sigma$. The equality $\Sigma_i = C_j^{\mathrm{ini}}.\Sigma$ makes the reification of delimited continuations well typed. When the handled term M evaluates to, say, $\sigma_j(V; y.\, N)$, the delimited continuation $\lambda y.$with H handle N is reified. Because the type of the continuation's body is C_j^{ini}, the *final* answer type of the term N handled in the continuation is C_j^{ini}. Therefore, as $\Sigma_i = C^{\mathrm{fin}}.\Sigma$ is required, $\Sigma_i = C_j^{\mathrm{ini}}.\Sigma$ has to hold.

3.4 Properties

First, we show that $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ is more expressive than both GEPCF and $\mathsf{HEPCF}^{\mathsf{ATM}}$. Note that the program syntax of the former subsumes those of the latter.

Theorem 1 ($\mathsf{GEPCF} \cup \mathsf{HEPCF}^{\mathsf{ATM}} \subset \mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$). *If M is well typed in either* GEPCF *or* $\mathsf{HEPCF}^{\mathsf{ATM}}$*, then it is also well typed in* $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$*. Furthermore, there exists a term M that is accepted by* $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ *but neither by* GEPCF *nor* $\mathsf{HEPCF}^{\mathsf{ATM}}$*.*

Type soundness of $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ is proven via progress and subject reduction [63].

Lemma 1 (Progress)**.** If $\emptyset \vdash M : C$, then one of the following holds: $M = \mathsf{return}\ V$ for some V; $M = \sigma(V; x.\, N)$ for some σ, V, x, and N; or $M \longrightarrow N$ for some N.

Lemma 2 (Subject Reduction)**.** If $\Gamma \vdash M : C$ and $M \longrightarrow N$, then $\Gamma \vdash N : C$.

We assume that a program is closed and handles all the ATM algebraic operations, as their behavior relies on captured delimited continuations. The latter condition is enforced by requiring the program's operation signature to take the form $\{\sigma_i : T_i^{\mathrm{par}} \rightsquigarrow T_i^{\mathrm{ari}} / \square\}^{1 \le i \le n}$. The remaining algebraic operations are considered as *primitive effects*, whose interpretations are given by, e.g., equational axioms on them [45].

Theorem 2 (Type Soundness). *Assume that $\Sigma = \{\sigma_i : T_i^{\mathrm{par}} \rightsquigarrow T_i^{\mathrm{ari}} / \square\}^{1 \le i \le n}$. If $\emptyset \vdash M : \Sigma \triangleright T / A$ and $M \longrightarrow^* N$ and $N \not\longrightarrow$, then either of the following holds:*

- *$N = \mathsf{return}\ V$ and $\emptyset \vdash V : T$ for some V; or*
- *$N = \sigma_i(V; x.\, L)$ and $\emptyset \vdash V : T_i^{\mathrm{par}}$ and $x : T_i^{\mathrm{ari}} \vdash L : \Sigma \triangleright T / A$ for some $i \in [1, n]$, V, x, and L.*

Note that, if $\Sigma = \emptyset$, it is ensured that all the operation calls performed at run time are handled.

4 Higher-Order Model Checking

The HOMC problem for $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ is defined using *effect trees* as the structures to be verified, and *alternating parity tree automata* (APTAs), which specify the semantics of primitive effects—corresponding to unhandled operations in our setting—and the properties to be verified. Due to lack of space, the paper only defines the *effect tree semantics*, which gives a way to interpret $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ terms as effect trees. Readers interested in the definition of APTAs and formal instances of the HOMC problem are referred to either the supplementary material or the literature [14, 29, 42, 56].

Effect trees are built by tree constructors labeled with (unhandled) algebraic operations or $\bot$ representing divergence. They are defined for closed $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ terms with *top-level* operation

signatures, which restrict algebraic operations with finitary parameter and arity types. To define effect trees, we use the following notation:

$$\begin{aligned} M \longrightarrow^n N &\stackrel{\text{def}}{=} \exists L_0, \cdots, L_n.\, M = L_0 \wedge (\forall\, i < n.\, L_i \longrightarrow L_{i+1}) \wedge L_n = N, \\ M \longrightarrow^* N &\stackrel{\text{def}}{=} \exists n.\, M \longrightarrow^n N, \text{ and} \\ M \longrightarrow^\omega &\stackrel{\text{def}}{=} \forall n.\, \exists N.\, M \longrightarrow^n N. \end{aligned}$$

Definition 1 (Tree Constructor Signatures)**.** A tree constructor signature S is a map from tree constructors, which are symbols ranged over by s, to natural numbers that represent the arities of the constructors. We write $S(s)$ for the arity of s assigned by S.

Definition 2 (Finitely Branching Infinite Trees)**.** The set $\mathbf{Tree}_S$ of finitely branching (possibly) infinite trees (or trees for short) generated by a tree constructor signature S is defined coinductively by the following grammar (where s is in the domain of S):

$$t \quad ::= \quad \bot \mid s(t_1, \cdots, t_{S(s)})\ .$$

Definition 3 (Effect Trees for $\mathsf{HEPCF}_\square^{\mathsf{ATM}}$ Terms)**.** An operation signature Σ is *top-level* if Σ takes the form $\{\sigma_i : B_i \rightsquigarrow E_i \,/\, \square\}^{1 \le i \le n}$. Given a top-level operation signature Σ and a type T, the tree constructor signature S_T^Σ is defined as follows:

$$S_T^\Sigma \stackrel{\text{def}}{=} \{(\sigma, n+1) \mid \sigma : B \rightsquigarrow \mathsf{n} \,/\, \square \in \Sigma\} \cup \{(\mathsf{return}\ V, 0) \mid \emptyset \vdash V : T\} \cup \bigcup_c \{(c, 0)\}\ ,$$

where a tree constructor is an algebraic operation, value-return construct, or base constant. Given a term M such that $\emptyset \vdash M : \Sigma \triangleright T \,/\, A$ with a top-level operation signature Σ, the effect tree of M, denoted by $\mathbf{ET}(M)$, is a tree in $\mathbf{Tree}_{S_T^\Sigma}$ defined coinductively as follows:

- if $M \longrightarrow^\omega$, then $\mathbf{ET}(M) = \bot$;
- if $M \longrightarrow^* \mathsf{return}\ V$, then $\mathbf{ET}(M) = \mathsf{return}\ V$; and
- if $M \longrightarrow^* \sigma(c; x.\, N)$ and $\sigma : B \rightsquigarrow \mathsf{n} \,/\, \square \in \Sigma$, then $\mathbf{ET}(M) = \sigma(c, \mathbf{ET}(N[\underline{1}/x]), \cdots, \mathbf{ET}(N[\underline{\mathsf{n}}/x]))$.

It is easy to confirm that, e.g., the effect tree semantic transforms the program $M_{\mathbf{File}}$ in Section 2 to the effect tree drawn there (with a slight modification to add parameters to operation nodes).

Finally, we define the HOMC problem as follows. A type T is *ground* if it is a base or enum type.

Definition 4 (Model Checking Problem for $\mathsf{HEPCF}_\square^{\mathsf{ATM}}$)**.** Given an APTA and a term M such that $\emptyset \vdash M : \Sigma \triangleright T \,/\, A$ for some top-level Σ and ground T, is $\mathbf{ET}(M)$ accepted by the APTA?

5 CPS Transformation

This section defines a selective, type-directed CPS transformation from $\mathsf{HEPCF}_\square^{\mathsf{ATM}}$ to EPCF and shows the decidability of the HOMC problem in $\mathsf{HEPCF}_\square^{\mathsf{ATM}}$ using it. We first recap EPCF briefly and then introduce the CPS transformation. Finally, we discuss the properties of the CPS transformation and the decidability proof.

5.1 Target Calculus EPCF

The calculus EPCF is a finitary variant of fine-grained call-by-value PCF with algebraic operations. We only show its syntax in Figure 6; see the supplementary material for the full definition. EPCF is similar to $\mathsf{HEPCF}_\square^{\mathsf{ATM}}$, except that EPCF does *not* support effect handlers (thus, answer-type modification does not appear in its type system either), types of values and terms are unified, and type signatures of algebraic operations are restricted to be in the form $\sigma : B \rightsquigarrow E$. The operational semantics and type system, equipped with typing judgments of the form $\Xi \parallel \Delta \vdash e : \tau$, are defined straightforwardly, and the effect tree semantics and the HOMC problem for EPCF are formalized as

$$
\begin{array}{rrcl}
\textbf{Values} & v, w & ::= & x \mid c \mid \varepsilon \mid \lambda x.e \mid \mathsf{fix}\, x.v \\
\textbf{Terms} & e & ::= & \mathsf{return}\, v \mid \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \mid v_1\, v_2 \mid \mathsf{case}(v; e_1, \cdots, e_n) \mid \sigma(v; x.\, e) \\
\textbf{Types} & \tau & ::= & B \mid E \mid \tau_1 \to \tau_2 \\
\textbf{Operation signatures} & \Xi & ::= & \{\sigma_i : B_i \rightsquigarrow E_i\}^{1 \le i \le n} \\
\textbf{Typing contexts} & \Delta & ::= & \emptyset \mid \Delta, x : \tau
\end{array}
$$

Fig. 6. Syntax of EPCF.

in Section 4. We write $\mathbf{ET}(e)$ for the effect tree of the EPCF term e. We only state the key property of EPCF: the HOMC problem is decidable. A type τ is ground if τ is a base or enum type.

Theorem 3 (Decidability of Model Checking for EPCF [14]). *Given an APTA and a term e such that $\Xi \parallel \emptyset \vdash e : \tau$ for some operation signature Ξ and ground type τ, the problem of checking whether $\mathbf{ET}(e)$ is accepted by the APTA is decidable.*

For readability, we use the following shorthand:

- Given $\overline{x} = x_1, \cdots, x_n$, we write $\lambda \overline{x}.e$ for the EPCF term $\lambda x_1.\mathsf{return}\, \lambda x_2.(\cdots (\mathsf{return}\, \lambda x_n.e)\ \cdots)$.
- Let X denote an EPCF term or value. Given an EPCF value v and $\overline{X}^{1 \le i \le n}$ $(n > 0)$, we write $v\, \overline{X}^{1 \le i \le n}$ for the EPCF term defined as follows:

$$
\begin{array}{llll}
v\, w \stackrel{\text{def}}{=} v\, w & & v\, (w, \overline{X}) \stackrel{\text{def}}{=} \mathsf{let}\, x = v\, w\, \mathsf{in}\, x\, \overline{X} & (\text{if } |\overline{X}| > 0) \\
v\, e \stackrel{\text{def}}{=} \mathsf{let}\, x = e\, \mathsf{in}\, v\, x & & v\, (e, \overline{X}) \stackrel{\text{def}}{=} \mathsf{let}\, x = e\, \mathsf{in}\, v\, (x\, \overline{X}) & (\text{if } |\overline{X}| > 0)
\end{array}
$$

  where the variable x is assumed to be fresh. Similarly, given a term e and $\overline{X}^{1 \le i \le n}$ $(n > 0)$, $e\, \overline{X}^{1 \le i \le n}$ means the EPCF term $\mathsf{let}\, x = e\, \mathsf{in}\, x\, \overline{X}^{1 \le i \le n}$ for some fresh variable x.

5.2 CPS Transformation

In this section, we first present an overview of the challenges one encounters when giving a CPS transformation for $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$, and of how they can be solved. After that, we define CPS transformation for types and programs in $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ and then show its properties.

5.2.1 Overview. We begin by reviewing a non-selective CPS transformation for effect handlers as presented in the literature [14, 22, 56], then identify the challenges in adapting it to our setting, and present our solution. Finally, we explain how this transformation is extended to a selective one.

CPS Transformation for Effect Handlers. In CPS-transforming terms with effect handlers, the transformation result takes two arguments: handlers and continuations. For instance, a value-return construct return V is transformed as follows:

$$[\![\mathsf{return}\, V]\!] \stackrel{\text{def}}{=} \lambda \overline{h}, k.k\, [\![V]\!]$$

where $\overline{h}$ are variables for handlers and k is for continuations. We call the pair consisting of zero or more handlers and a continuation a *contextual argument*. Because the value-return construct does not call algebraic operations, it only passes the CPS value $[\![V]\!]$ to the continuation. The handler variables are used to transform operation calls. For instance, an operation call $\sigma(V; x.\, M)$ is transformed as:

$$[\![\sigma(V; x.\, M)]\!] \stackrel{\text{def}}{=} \lambda \overline{h}, k.h^{\sigma}\, [\![V]\!]\, \lambda x.[\![M]\!]\, \overline{h}\, k$$

where h^{σ} is the variable in $\overline{h}$ and represents the clause to handle the algebraic operation σ. Given the clause, it takes the CPS parameter value $[\![V]\!]$ and the delimited continuation of the form $\lambda x.[\![M]\!]\, \overline{h}\, k$.

Typechecking Issue. However, an issue with typechecking arises in the CPS transformation of handling constructs. Following the prior work [22], the transformation can be defined as follows:

$$[\![\mathsf{with}\ H\ \mathsf{handle}\ M]\!] \stackrel{\mathrm{def}}{=} \lambda \overline{h}, k.[\![M]\!]\ \overline{v^{\mathsf{h}}}\ v^{\mathsf{k}}\ \overline{h}\ k \tag{1}$$

where $\overline{v^{\mathsf{h}}}$ and v^{k} are transformation results of the operation clauses and the return clause, respectively, in the effect handler H. It is noteworthy that, with this definition, a CPS-transformed term may take multiple contextual arguments, but their number *depends on the context*. In general, given a source term with H_1 handle $(\cdots\ (\mathsf{with}\ H_n\ \mathsf{handle}\ M)\ \cdots)$, $[\![M]\!]$ takes at least n contextual arguments. To ensure that the CPS transformation preserves typing, we require a way to typecheck CPS terms that take multiple contextual arguments.

There are three ways to address this issue in the literature. The first solution is to use *parametric polymorphism* [22], which allows parameterizing CPS terms over contextual arguments and instantiating when necessary. However, since parametric polymorphism makes the HOMC problem undecidable [62], it is inadequate for our purpose. The second is to adopt an ATM type system for the source language [56]. Because the ATM type system exposes the dependency of terms on the context, it can statically capture how many contextual arguments CPS terms depend on. Based on this idea, Sekiyama and Unno gave a CPS transformation for $\mathsf{HEPCF}^{\mathsf{ATM}}$. The third is to fix the number of contextual arguments CPS terms take by restricting effect handlers to be tail-resumptive [14]. In this solution, the CPS transformation of handling constructs is defined to be

$$[\![\mathsf{with}\ H\ \mathsf{handle}\ M]\!] \stackrel{\mathrm{def}}{=} \lambda \overline{h}, k.[\![M]\!]\ \overline{v^{\mathsf{h}}}\ v^{\mathsf{k}}$$

where the handlers $\overline{h}$ and continuation k are "weaved" into $\overline{v^{\mathsf{h}}}$ and v^{k}. Namely, for each tail-resumptive clause $\sigma(x) \mapsto M \in H$, $\overline{v^{\mathsf{h}}}$ gives a CPS value $\lambda x, k'.[\![M]\!]\ \overline{h}\ k'$, and, given the clause $\mathsf{return}\ x \mapsto N \in H$, v^{k} is defined to be $\lambda x.[\![N]\!]\ \overline{h}\ k$. This transformation of handlers and continuations preserves the semantics of the original handling construct *if* the effect handler only includes tail-resumptive operation clauses, but it does not in general. For instance, if the effect handler H were to allow an operation clause to return some value without invoking the captured continuation (such as exception handling), then the continuation k passed to the handling construct would be discarded, while its correct semantics is that the result of evaluating the clause is passed to the continuation k. Under the assumption that all the operation clauses are tail-resumptive, no problem happens because, if the evaluation with H handle M terminates at some value, the CPS term for M finally invokes the continuation v^{k}, which in turn invokes the outer continuation k.

Our Solution. Our approach to the issue with handling constructs is to mix the second and third solutions. That is, we adopt transformation (1) but we only weave the outer handler $\overline{h}$ into the CPS values for tail-resumptive operation clauses and track the contextual arguments taken by the other clauses using the ATM type system of $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$.

However, the naive mix causes semantic unsoundness. For example, suppose that the handled term M is an operation call $\sigma(V; x.N)$ and the effect handler H includes a clause $\sigma(y; k^{\sigma}) \mapsto \mathsf{with}\ H'\ \mathsf{handle}\ k^{\sigma}\ y$ with another effect handler H'. Then, the fully applied CPS value for the term with H handle M with outer handler $\overline{v_0^{\mathsf{h}}}$ and continuation v_0^{k} is evaluated as follows:

$$\begin{aligned}
[\![\mathsf{with}\ H\ \mathsf{handle}\ \sigma(V; x.N)]\!]\ \overline{v_0^{\mathsf{h}}}\ v_0^{\mathsf{k}} \quad &= \quad (\lambda \overline{h}, k.[\![\sigma(V; x.N)]\!]\ \overline{v^{\mathsf{h}}}\ v^{\mathsf{k}}\ \overline{h}\ k)\ \overline{v_0^{\mathsf{h}}}\ v_0^{\mathsf{k}} \\
&\longrightarrow^* \quad [\![\sigma(V; x.N)]\!]\ \overline{v^{\mathsf{h}}[\overline{v_0^{\mathsf{h}}}/\overline{h}]}\ v^{\mathsf{k}}\ \overline{v_0^{\mathsf{h}}}\ v_0^{\mathsf{k}} \\
&\longrightarrow^* \quad (v^{\sigma}\ [\![V]\!]\ (\lambda x.[\![N]\!]\ \overline{v^{\mathsf{h}}[\overline{v_0^{\mathsf{h}}}/\overline{h}]}\ v^{\mathsf{k}}))\ \overline{v_0^{\mathsf{h}}}\ v_0^{\mathsf{k}}
\end{aligned}$$

where $\overline{v^{\mathsf{h}}[\overline{v_0^{\mathsf{h}}}/\overline{h}]}$ means that each variable in $\overline{h}$ occurring in $\overline{v^{\mathsf{h}}}$ is replaced by the corresponding value in $\overline{v_0^{\mathsf{h}}}$ (recall that the CPS values for tail-resumptive clauses refer to $\overline{h}$), and v^σ is the CPS value for the clause of σ, which can be given

$$v^\sigma = \lambda y, k^\sigma.\mathsf{return}\ [\![\mathsf{with}\ H'\ \mathsf{handle}\ k^\sigma\ y]\!]\ .$$

Therefore, the continuation $\lambda x.[\![N]\!]\ \overline{v^{\mathsf{h}}[\overline{v_0^{\mathsf{h}}}/\overline{h}]}\ v^{\mathsf{k}}$ passed to the operation clause is invoked under another effect handler H'. Assume that the term N calls an algebraic operation ς and the effect handler H involves an operation clause $\varsigma(z) \mapsto \varsigma(z)$, i.e., the operation call is forwarded into the outer context. According to the semantics of the source language, the forwarded operation call is handled by H' because it is the closest effect handler from the forwarding. However, in the given CPS term, *the forwarded operation call is handled by the outer effect handler* $\overline{v_0^{\mathsf{h}}}$, as the handler $\overline{v^{\mathsf{h}}[\overline{v_0^{\mathsf{h}}}/\overline{h}]}$, where operation calls forwarded by tail-resumptive clauses are interpreted by $\overline{v_0^{\mathsf{h}}}$, is passed to $[\![N]\!]$. In summary, only weaving the outer handler $\overline{v_0^{\mathsf{h}}}$ into tail-resumptive clauses does not work when a captured continuation is invoked under a different effect handler. Note that such a phenomenon does not happen when all the operation clauses are tail-resumptive [14].

We solve this issue by making *continuations take tail-resumptive clauses*. In our CPS transformation, an operation call $\sigma(V; x.\, N)$ is transformed into

$$\lambda\overline{h}, k.v^\sigma\ [\![V]\!]\ (\lambda x, \overline{h_0}.N\ \overline{h'}\ \overline{h_0}\ k)$$

where $\overline{h'}$ is a subsequence of $\overline{h}$ that only gathers ATM clauses, and $\overline{h_0}$ is a sequence of tail-resumptive ones given by the call site of the continuation. Then, for the above problematic example, we have

$$\begin{aligned}[\![\mathsf{with}\ H\ \mathsf{handle}\ \sigma(V; x.\, N)]\!]\ \overline{v_0^{\mathsf{h}}}\ v_0^{\mathsf{k}} \quad &\longrightarrow^* \quad [\![\sigma(V; x.\, N)]\!]\ \overline{v^{\mathsf{h}}[\overline{v_0^{\mathsf{h}}}/\overline{h}]}\ v^{\mathsf{k}}\ \overline{v_0^{\mathsf{h}}}\ v_0^{\mathsf{k}} \\ &\longrightarrow^* \quad (v^\sigma\ [\![V]\!]\ (\lambda x, \overline{h_0}.[\![N]\!]\ \overline{v^{\mathsf{h}'}}\ \overline{h_0}\ v^{\mathsf{k}}))\ \overline{v_0^{\mathsf{h}}}\ v_0^{\mathsf{k}}\end{aligned}$$

where $\overline{v^{\mathsf{h}'}}$ is the sequence of the ATM clauses in $\overline{v^{\mathsf{h}}}$, so it is *independent of the outer handler* $\overline{v_0^{\mathsf{h}}}$.

The remaining challenge is how to pass, for $\overline{h_0}$, the tail-resumptive clauses in H that weave the effect handler at the call site of the continuation. To resolve it, we modify the definition of the CPS value v^σ as follows:

$$v^\sigma \stackrel{\mathrm{def}}{=} \lambda y, k'^\sigma.\mathsf{let}\ k^\sigma = \mathsf{return}\ \lambda z, \overline{h}, k.k'^\sigma\ z\ \overline{v}\ \overline{h}\ k\ \mathsf{in}\ \mathsf{return}\ [\![\mathsf{with}\ H'\ \mathsf{handle}\ k^\sigma\ y]\!]\ , \tag{2}$$

where the continuation k^σ used in the body is a wrapper of the given continuation k'^σ. The wrapper takes an argument z, handler $\overline{h}$, and continuation k from the call-site and weaves $\overline{h}$ into the tail-resumptive clauses in H (the weaving results are referred to by $\overline{v}$ in Definition (2)), and then passes them to the given continuation k'^σ. Given a tail-resumptive clause $\varsigma(z) \mapsto L$ in H, the weaving result in $\overline{v}$ is given by $\lambda z, k'.[\![L]\!]\ \overline{h}\ k'$, which refers to the handler $\overline{h}$ at the call site.

Selective CPS Transformation. While we have focused on *non*-selective CPS transformations thus far, what we actually define is a *selective* CPS transformation, which transforms $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$ terms with ATM effects (resp. the ATP effect $\square$) into EPCF terms that do (resp. do not) require continuations. Thus, e.g., the transformation of value-return constructs is defined as follows:

$$[\![\mathsf{return}\ V]\!] \stackrel{\mathrm{def}}{=} \lambda\overline{h}, \_.\mathsf{return}\ [\![V]\!]\ ,$$

which takes a handler $\overline{h}$ and an extra, unused argument _. We can define a transformation that does not take the extra argument, but leaving it enables giving a simpler CPS transformation that unifies the transformations for ATP- and ATM-effectful terms. For the ATP-effectful terms, the

$\boxed{[\![T]\!]}$ **for value types**

$$[\![B]\!] \stackrel{\text{def}}{=} B \qquad [\![E]\!] \stackrel{\text{def}}{=} E \qquad [\![T \to C]\!] \stackrel{\text{def}}{=} [\![T]\!] \to [\![C]\!]$$

$\boxed{[\![C]\!]}$ **for computation types**

$$[\![\Sigma \triangleright T \,/\, \square]\!] \stackrel{\text{def}}{=} [\![\Sigma]\!][\, 1 \to [\![T]\!] \,]$$
$$[\![\Sigma \triangleright T \,/\, C^{\text{ini}} \Rightarrow C^{\text{fin}}]\!] \stackrel{\text{def}}{=} [\![\Sigma]\!][\, ([\![T]\!] \to [\![\square(\Sigma)]\!][\, [\![C^{\text{ini}}]\!] \,]) \to [\![C^{\text{fin}}]\!] \,]$$

$\boxed{[\![\Sigma]\!][\, \tau \,]}$ **for operation signatures**

$$[\![\{\sigma_i : T_i^{\text{par}} \rightsquigarrow T_i^{\text{ari}} \,/\, C_i^{\text{ini}} \Rightarrow C_i^{\text{fin}}\}^{1\le i\le m} \uplus \{\varsigma_i : U_i^{\text{par}} \rightsquigarrow U_i^{\text{ari}} \,/\, \square\}^{1\le i\le n}]\!][\, \tau \,] \stackrel{\text{def}}{=}$$
$$\overline{([\![T_i^{\text{par}}]\!] \to ([\![T_i^{\text{ari}}]\!] \to \overline{\tau_j \to}^{1\le j\le n} [\![C_i^{\text{ini}}]\!]) \to [\![C_i^{\text{fin}}]\!])}^{1\le i\le m} \overline{\tau_i \to}^{1\le i\le n} \tau$$
$$(\text{where } \forall\, i \in [1,n].\ \tau_i = [\![U_i^{\text{par}}]\!] \to [\![U_i^{\text{ari}}]\!])$$

Fig. 7. CPS transformation for types.

enum constant $\underline{1}$ is given instead of continuations. For example, a let-construct $\mathsf{let}\, x = M_1 \,\mathsf{in}\, M_2$ where the control effect of M_1 is $\square$, is transformed into

$$\lambda \overline{h}, k.\mathsf{let}\, x = [\![M_1]\!]\, \overline{h}\, \underline{1}\, \mathsf{in}\, [\![M_2]\!]\, \overline{h}\, k\,,$$

where the result of $[\![M_1]\!]$ is bound to x because $[\![M_1]\!]$ does not take continuations. The variable k represents the enum constant or a continuation, depending on the control effect of M_2.

5.2.2 Definition. The CPS transformation for types is defined in Figure 7 using the notation $\overline{\tau_i \to}^{1\le i\le n} \tau$, which stands for $\tau_1 \to \tau_2 \to \cdots \to \tau_n \to \tau$ (when $n = 0$, it denotes τ). Value types are transformed in a standard manner. An operation signature Σ is transformed into a function $[\![\Sigma]\!][-]$ that wraps a given type to take a CPS-transformed handler conforming to Σ. The definition indicates that each ATP operation $\varsigma : U^{\text{par}} \rightsquigarrow U^{\text{ari}} \,/\, \square$ is transformed into a value of $[\![U^{\text{par}}]\!] \to [\![U^{\text{ari}}]\!]$, which does not take continuations. The CPS value of an ATM operation $\sigma : T^{\text{par}} \rightsquigarrow T^{\text{ari}} \,/\, C^{\text{ini}} \Rightarrow C^{\text{fin}}$ takes a parameter of $[\![T^{\text{par}}]\!]$ and a continuation, and then returns a CPS value of the final answer type $[\![C^{\text{fin}}]\!]$. The continuation takes a value of the arity type $[\![T^{\text{ari}}]\!]$ (which represents the return value of the call to σ) along with tail-resumptive clauses weaving the call-site's effect handler and returns a CPS value of the initial answer type $[\![C^{\text{ini}}]\!]$. The definition for computation types indicates that an $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ term is transformed into a CPS term taking a handler. If the control effect of the $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ term is $\square$, then the CPS term only returns a CPS value that is the result of the computation. Otherwise, the CPS term takes a continuation and returns a "final answer." Here, $\square(\Sigma)$ is an operation signature that only gathers ATP operations in Σ. Thus, the type $[\![T]\!] \to [\![\square(\Sigma)]\!][\, [\![C^{\text{ini}}]\!] \,]$ means that the continuation takes tail-resumptive clauses. We write $[\![\Gamma]\!]$ for the EPCF typing context obtained by CPS-transforming the types of all the bindings of typing context Γ.

To define CPS transformation for values and terms, we introduce the *static lambda calculus*, which allows us to remove *administrative redexes* inserted at the time of CPS transformation.

Definition 5 (Static Lambda Calculus)**.** The static lambda calculus (SLC) is defined by:

$$\mathbf{t} \stackrel{\text{def}}{=} \mathbf{x} \mid \boldsymbol{\lambda}(\mathbf{x}_1, \cdots, \mathbf{x}_n).\, \mathbf{t} \mid \mathbf{t}@(\mathbf{t}_1, \cdots, \mathbf{t}_n) \mid e \mid v$$

$\boxed{[\![V]\!]}$ **for values**

$$[\![x]\!]_{\text{HT\_Var}} \stackrel{\text{def}}{=} x \qquad [\![c]\!]_{\text{HT\_Const}} \stackrel{\text{def}}{=} c \qquad [\![\underline{n}]\!]_{\text{HT\_EConst}} \stackrel{\text{def}}{=} \underline{n}$$
$$[\![\lambda x.M]\!]_{\text{HT\_Abs}} \stackrel{\text{def}}{=} \lambda x.\mathsf{return}\,[\![M]\!] \qquad [\![\mathsf{fix}\,x.V]\!]_{\text{HT\_Fix}} \stackrel{\text{def}}{=} \mathsf{fix}\,x.[\![V]\!]$$

$\boxed{[\![M]\!]}$ **for thunks** (assume that $\Gamma \vdash M : \Sigma \triangleright T \,/\, A$ and $|\overline{h}| = |\Sigma|$)

$$[\![M]\!] \stackrel{\text{def}}{=} \lambda\overline{h}, k.[\![M]\!]^{\mathrm{e}}@(\overline{h}, k)$$

$\boxed{[\![M]\!]^{\mathrm{e}}}$ **for terms** (assume $\Gamma \vdash M : \Sigma \triangleright T \,/\, A$ and $|\overline{\mathbf{h}}| = |\Sigma|$ and $|\overline{\mathbf{h}^{\not\square}}| = |\not\square(\Sigma)|$ and $|\overline{\mathbf{h}^{\square}}| = |\overline{h^{\square}}| = |\square(\Sigma)|$)

$$[\![\mathsf{return}\,V]\!]^{\mathrm{e}}_{\text{HT\_Return}} \stackrel{\text{def}}{=} \boldsymbol{\lambda}(\overline{\mathbf{h}}, \mathbf{k}).\,\mathsf{return}\,[\![V]\!]$$
$$[\![M]\!]^{\mathrm{e}}_{\text{HT\_Emb}} \stackrel{\text{def}}{=} \boldsymbol{\lambda}(\overline{\mathbf{h}}, \mathbf{k}).\,\mathsf{let}\,x = [\![M]\!]^{\mathrm{e}}@(\overline{\mathbf{h}}, \underline{1})\,\mathsf{in}\,\mathbf{k}\,x$$
$$[\![\mathsf{let}\,x = M\,\mathsf{in}\,N]\!]^{\mathrm{e}}_{\text{HT\_Let}} \stackrel{\text{def}}{=} \boldsymbol{\lambda}(\overline{\mathbf{h}}, \mathbf{k}).\,\mathsf{let}\,x = [\![M]\!]^{\mathrm{e}}@(\overline{\mathbf{h}}, \underline{1})\,\mathsf{in}\,[\![N]\!]^{\mathrm{e}}@(\overline{\mathbf{h}}, \mathbf{k})$$
$$[\![\mathsf{let}\,x = M\,\mathsf{in}\,N]\!]^{\mathrm{e}}_{\text{HT\_LetATM}} \stackrel{\text{def}}{=} \boldsymbol{\lambda}(\overline{\mathbf{h}}, \mathbf{k}).\,[\![M]\!]^{\mathrm{e}}@(\overline{\mathbf{h}}, \lambda x, \overline{h^{\square}}.[\![N]\!]^{\mathrm{e}}@(\overline{\mathbf{h}^{\not\square}}, \overline{h^{\square}}, \mathbf{k}))$$
$$[\![V\ W]\!]^{\mathrm{e}}_{\text{HT\_App}} \stackrel{\text{def}}{=} \boldsymbol{\lambda}(\overline{\mathbf{h}}, \mathbf{k}).\,[\![V]\!]\,[\![W]\!]\,\overline{\mathbf{h}}\,\mathbf{k}$$
$$[\![\mathsf{case}(V; M_1, \cdots, M_n)]\!]^{\mathrm{e}}_{\text{HT\_Case}} \stackrel{\text{def}}{=} \boldsymbol{\lambda}(\overline{\mathbf{h}}, \mathbf{k}).\,\mathsf{case}([\![V]\!]; [\![M_1]\!]^{\mathrm{e}}@(\overline{\mathbf{h}}, \mathbf{k}), \cdots, [\![M_n]\!]^{\mathrm{e}}@(\overline{\mathbf{h}}, \mathbf{k}))$$
$$[\![\sigma(V; x.\,M)]\!]^{\mathrm{e}}_{\text{HT\_Op}} \stackrel{\text{def}}{=} \boldsymbol{\lambda}(\overline{\mathbf{h}}, \mathbf{k}).\,\mathsf{let}\,x = \mathbf{h}^{\sigma}\,[\![V]\!]\,\mathsf{in}\,[\![M]\!]^{\mathrm{e}}@(\overline{\mathbf{h}}, \mathbf{k})$$
$$[\![\sigma(V; x.\,M)]\!]^{\mathrm{e}}_{\text{HT\_OpATM}} \stackrel{\text{def}}{=} \boldsymbol{\lambda}(\overline{\mathbf{h}}, \mathbf{k}).\,\mathbf{h}^{\sigma}\,[\![V]\!]\,\lambda x, \overline{h^{\square}}.[\![M]\!]^{\mathrm{e}}@(\overline{\mathbf{h}^{\not\square}}, \overline{h^{\square}}, \mathbf{k})$$
$$[\![\mathsf{with}\,H\,\mathsf{handle}\,M]\!]^{\mathrm{e}}_{\text{HT\_Handle}} \stackrel{\text{def}}{=} \boldsymbol{\lambda}(\overline{\mathbf{h}}, \mathbf{k}).\,[\![M]\!]^{\mathrm{e}}@(\overline{w_i^{\not\square}}^{1\le i\le m}, \overline{w_i^{\square}}^{1\le i\le n}, \lambda x, \overline{h_i}^{1\le i\le n}.\mathsf{return}\,[\![L]\!])\,\overline{\mathbf{h}}\,\mathbf{k}\,,\ \text{where}$$

$$H = \{\mathsf{return}\,x \mapsto L\} \uplus \{\sigma_i(y_i; k_i) \mapsto M_i\}^{1\le i\le m} \uplus \{\varsigma_i(z_i) \mapsto N_i\}^{1\le i\le n}$$
$$\forall i \in [1,m].\,\Gamma, y_i : T_i^{\mathrm{par}}, k_i : T_i^{\mathrm{ari}} \to C_i^{\mathrm{ini}} \vdash M_i : C_i^{\mathrm{fin}}$$
$$\forall i \in [1,m].\,w_i^{\not\square} \stackrel{\text{def}}{=} \lambda y_i, k_i'.\mathsf{let}\,k_i = \mathsf{return}\,(\lambda y, \overline{h}, k.k_i'\,y\,\overline{v_{ij}}^{1\le j\le n}\,\overline{h}\,k)\,\mathsf{in}\,\mathsf{return}\,[\![M_i]\!] \quad (|\overline{h}| = |C_i^{\mathrm{ini}}.\Sigma|)$$
$$\forall i \in [1,m], j \in [1,n].\,v_{ij} \stackrel{\text{def}}{=} \lambda z_j.[\![N_j]\!]^{\mathrm{e}}@(\overline{h}, \underline{1}) \quad (|\overline{h}| = |C_i^{\mathrm{ini}}.\Sigma|)$$
$$\forall i \in [1,n].\,w_i^{\square} \stackrel{\text{def}}{=} \lambda z_i.[\![N_i]\!]^{\mathrm{e}}@(\overline{\mathbf{h}}, \underline{1})$$

Fig. 8. CPS transformation for terms.

where e and v are EPCF terms and values, respectively, that may refer to static variables $\mathbf{x}$ bounded in the enclosing context. We call variables, functions, and applications in the SLC *static*. A static application $(\boldsymbol{\lambda}(\mathbf{x}_1, \cdots, \mathbf{x}_n).\,\mathbf{t})@(\mathbf{t}_1, \cdots, \mathbf{t}_n)$ is identified with the β-reduction result $\mathbf{t}[\mathbf{x}_1 := \mathbf{t}_1, \cdots, \mathbf{x}_n := \mathbf{t}_n]$.

For $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$ terms and values, we define three kinds of CPS transformation: $[\![V]\!]$ for values, $[\![M]\!]$ and $[\![M]\!]^{\mathrm{e}}$ for terms. These CPS transformations are defined on typing derivations of values and terms, not on values and terms themselves. For clarification, we may specify the name of the typing rule that is used lastly for constructing a typing derivation to be transformed, as $[\![M]\!]^{\mathrm{e}}_{\text{HT\_Let}}$, which is the result of transforming a typing derivation that concludes $\Gamma \vdash M : C$ for some Γ and C using (HT_Let). The definition of the CPS transformation is presented in Figure 8. The CPS transformation $[\![V]\!]$ for a value V is defined in a homomorhpic manner, and $[\![M]\!]$ is an EPCF function that takes a contextual argument (a handler and a continuation) and statically applies the static function $[\![M]\!]^{\mathrm{e}}$ to it. The CPS transformation $[\![M]\!]^{\mathrm{e}}$ for a term M is a static function that maps a contextual argument to an EPCF term, defined as explained in Section 5.2.1. We only describe a few remarks here. First, given $\overline{\mathbf{h}}$, we write $\overline{\mathbf{h}^{\not\square}}$ for the subsequence of $\overline{\mathbf{h}}$ that exclusively includes all the clauses for ATM algebraic operations. We also write $\mathbf{h}^{\sigma}$ for the clause of σ given by $\overline{\mathbf{h}}$. Note that the clause $\mathbf{h}^{\sigma}$ for an ATP operation σ takes no continuation.

5.2.3 Properties. We show that our CPS transformation preserves types and semantics of $\mathsf{HEPCF}^{\mathsf{ATM}}_{\Box}$ terms. We first introduce some notions used to formulate type preservation.

Definition 6 (Partial Order on EPCF Typing Contexts)**.** We write $\Delta_1 \preceq \Delta_2$ if $dom(\Delta_1) \subseteq dom(\Delta_2)$ and, for any $x \in dom(\Delta_1)$, $\Delta_1(x) = \Delta_2(x)$.

Definition 7 (Typing of Effect Handlers)**.** Let $\Sigma = \{\sigma_i : T_i^{\mathrm{par}} \rightsquigarrow T_i^{\mathrm{ari}} \,/\, C_i^{\mathrm{ini}} \Rightarrow C_i^{\mathrm{fin}}\}^{1\le i\le m} \uplus \{\varsigma_i : U_i^{\mathrm{par}} \rightsquigarrow U_i^{\mathrm{ari}} \,/\, \Box\}$. For a value sequence $\overline{v} = v^{\sigma_1}, \cdots, v^{\sigma_m}, v^{\varsigma_1}, \cdots, v^{\varsigma_n}$, we write $\Xi \parallel \Delta \vdash \overline{v} : \Sigma$ if (1) $\forall\, i \in [1,m].\ \Xi \parallel \Delta \vdash v^{\sigma_i} : [\![T_i^{\mathrm{par}}]\!] \to ([\![T_i^{\mathrm{ari}}]\!] \to [\![\Box(\Sigma)]\!][\,[\![C_i^{\mathrm{ini}}]\!]\,]) \to [\![C_i^{\mathrm{fin}}]\!]$ and (2) $\forall\, i \in [1,n].\ \Xi \parallel \Delta \vdash v^{\varsigma_i} : [\![U_i^{\mathrm{par}}]\!] \to [\![U_i^{\mathrm{ari}}]\!]$.

Definition 8 (Types of the Static Lambda Calculus)**.** We write (1) $\mathsf{term}[\,\Xi \parallel \Delta \vdash \tau\,]$ for the set of EPCF terms e such that $\Xi \parallel \Delta \vdash e : \tau$, (2) $\mathsf{val}[\,\Xi \parallel \Delta \vdash \tau\,]$ for the set of EPCF values v such that $\Xi \parallel \Delta \vdash v : \tau$, and (3) $\mathsf{vals}[\,\Xi \parallel \Delta \vdash \Sigma\,]$ for the set of sequences of EPCF values $\overline{v}$ such that $\Xi \parallel \Delta \vdash \overline{v} : \Sigma$. We also define $\mathsf{comp}[\,\Xi \parallel \Delta \vdash C\,]$ depending on $C.A$, as follows:

$$\mathsf{comp}[\,\Xi \parallel \Delta \vdash \Sigma \triangleright T \,/\, \Box\,] \stackrel{\mathrm{def}}{=} \mathsf{vals}[\,\Xi \parallel \Delta \vdash \Sigma\,] \times \mathsf{val}[\,\Xi \parallel \Delta \vdash 1\,] \to \mathsf{term}[\,\Xi \parallel \Delta \vdash [\![T]\!]\,]$$

$$\mathsf{comp}[\,\Xi \parallel \Delta \vdash \Sigma \triangleright T \,/\, C^{\mathrm{ini}} \Rightarrow C^{\mathrm{fin}}\,] \stackrel{\mathrm{def}}{=} \mathsf{vals}[\,\Xi \parallel \Delta \vdash \Sigma\,] \times (\mathsf{val}[\,\Xi \parallel \Delta \vdash [\![T]\!] \to [\![\Box(\Sigma)]\!][\,[\![C^{\mathrm{ini}}]\!]\,]\,]) \to \mathsf{term}[\,\Xi \parallel \Delta \vdash [\![C^{\mathrm{fin}}]\!]\,]$$

Lemma 3 (Type Preservation of the CPS Transformation)**.** Assume that $[\![\Gamma]\!] \preceq \Delta$.

(1) If $\Gamma \vdash V : T$, then $\Xi \parallel \Delta \vdash [\![V]\!] : [\![T]\!]$ for any Ξ.
(2) If $\Gamma \vdash M : C$, then $\Xi \parallel \Delta \vdash [\![M]\!] : [\![C]\!]$ for any Ξ.
(3) If $\Gamma \vdash M : C$, then $[\![M]\!]^{\mathsf{e}} : \mathsf{comp}[\,\Xi \parallel \Delta \vdash C\,]$ for any Ξ.

For semantic preservation, we first prove that the evaluation of $\mathsf{HEPCF}^{\mathsf{ATM}}_{\Box}$ terms can be simulated by their CPS terms. The simulation holds modulo *full $\beta\eta$ monadic reduction*.

Definition 9 (Full $\beta\eta$ Monadic Reduction)**.** We define a binary relation $\hookrightarrow$ over EPCF terms and over EPCF values, called *full $\beta\eta$ monadic reduction*, to be the reflexive, transitive, compatible closure satisfying the following axioms:

$$\forall\, x, v, e.\ (\lambda x.e)\ v \hookrightarrow e[v/x] \qquad \forall\, x, v.\ x \notin fv(v) \Longrightarrow \lambda x.v\,x \hookrightarrow v$$
$$\forall\, x, v, e.\ \mathsf{let}\ x = \mathsf{return}\ v\ \mathsf{in}\ e \hookrightarrow e[v/x] \qquad \forall\, x, e.\ \mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{return}\ x \hookrightarrow e$$
$$\forall\, x, y, e_1, e_2, e_3.\ y \notin fv(e_3) \Longrightarrow \mathsf{let}\ x = (\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2)\ \mathsf{in}\ e_3 \hookrightarrow \mathsf{let}\ y = e_1\ \mathsf{in}\ \mathsf{let}\ x = e_2\ \mathsf{in}\ e_3$$

Lemma 4 (Simulation up to Reduction)**.** If $\emptyset \vdash M : \Sigma \triangleright T \,/\, \Box$ and $|\overline{v^{\mathsf{h}}}| = |\Sigma|$, then, for any v^{k}, one of the following holds:

(1) $M = \mathsf{return}\ V$ and $[\![\mathsf{return}\ V]\!]^{\mathsf{e}}@(\overline{v^{\mathsf{h}}}, v^{\mathsf{k}}) = \mathsf{return}\ [\![V]\!]$ for some V;
(2) $M \longrightarrow^{*} \sigma(V'; x.\,M')$ and $[\![M]\!]^{\mathsf{e}}@(\overline{v^{\mathsf{h}}}, v^{\mathsf{k}}) \hookrightarrow \mathsf{let}\ x = v\,[\![V']\!]\ \mathsf{in}\ [\![M']\!]^{\mathsf{e}}@(\overline{v^{\mathsf{h}}}, v^{\mathsf{k}})$ for some σ, V', x, M', and v such that v is a value in the sequence $\overline{v^{\mathsf{h}}}$ that corresponds to σ; or
(3) $M \longrightarrow^{+} M'$ and $[\![M]\!]^{\mathsf{e}}@(\overline{v^{\mathsf{h}}}, v^{\mathsf{k}}) \hookrightarrow \longrightarrow^{+} \hookrightarrow [\![M']\!]^{\mathsf{e}}@(\overline{v^{\mathsf{h}}}, v^{\mathsf{k}})$ for some M'.

To show Lemma 4, we have to consider a more complicated statement because handled terms in handling constructs have ATM effects, while Lemma 4 only considers the ATP effect for simplification. See the supplementary material for the full statement. Using the bisimilarity technique of Dal Lago et al. [13], we can prove that $\hookrightarrow$ preserves *contextual improvement*, which is a partial order relating contextually equivalent terms e_1 and e_2 such that e_2 takes at least the same number of evaluation steps as e_1. By this property with Lemma 4, the *observational behavior* (termination at values/operation calls, or divergence) of $\mathsf{HEPCF}^{\mathsf{ATM}}_{\Box}$ terms can be simulated by their CPS terms. Note that, if the simulation modulo $\beta\eta$ monadic *equality* were to be proven, ensuring the simulation

Subtyping rules $\boxed{T_1 <: T_2}$ $\boxed{C_1 <: C_2}$ $\boxed{A_1 <: A_2}$ $\boxed{\Sigma_1 <: \Sigma_2}$

$$B <: B \qquad E <: E \qquad \frac{T_2 <: T_1 \quad C_1 <: C_2}{T_1 \to C_1 <: T_2 \to C_2}$$

$$\frac{\Sigma_2 <: \Sigma_1 \quad T_1 <: T_2 \quad A_1 <: A_2 \quad A_1 \neq \square \Longrightarrow \square(\Sigma_1) <: \square(\Sigma_2)}{\Sigma_1 \triangleright T_1 \,/\, A_1 <: \Sigma_2 \triangleright T_2 \,/\, A_2}$$

$$\square <: \square \qquad \frac{C_1 <: C_2}{\square <: C_1 \Rightarrow C_2} \qquad \frac{C_2^{\mathrm{ini}} <: C_1^{\mathrm{ini}} \quad C_1^{\mathrm{fin}} <: C_2^{\mathrm{fin}}}{C_1^{\mathrm{ini}} \Rightarrow C_1^{\mathrm{fin}} <: C_2^{\mathrm{ini}} \Rightarrow C_2^{\mathrm{fin}}}$$

$$\frac{\forall\, i \in [1,n].\ T_{2i}^{\mathrm{par}} <: T_{1i}^{\mathrm{par}} \ \wedge\ T_{1i}^{\mathrm{ari}} <: T_{2i}^{\mathrm{ari}} \ \wedge\ A_{1i} <: A_{2i}}{\{\sigma_i : T_{1i}^{\mathrm{par}} \rightsquigarrow T_{1i}^{\mathrm{ari}} \,/\, A_{1i}\}^{1\le i\le n} \uplus \Sigma <: \{\sigma_i : T_{2i}^{\mathrm{par}} \rightsquigarrow T_{2i}^{\mathrm{ari}} \,/\, A_{2i}\}^{1\le i\le n}}$$

Additional typing rules $\boxed{\Gamma \vdash V : T}$ $\boxed{\Gamma \vdash M : C}$

$$\frac{\Gamma \vdash V : T \quad T <: U}{\Gamma \vdash V : U} \qquad \frac{\Gamma \vdash M : C \quad C <: D}{\Gamma \vdash M : D}$$

Fig. 9. Subtyping.

of observational behavior—more specifically, proving that, if an $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ term diverges, its CPS term takes an infinitely many number of steps—would be more challenging. Now, we show that the CPS transformation preserves effect trees of given $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ terms using the above result.

Theorem 4 (Preservation of Effect Trees). *Let T be a ground type and $\Sigma = \{\sigma_i : B_i \rightsquigarrow E_i \,/\, \square\}^{1\le i\le n}$ and $\Xi = \{\sigma_i : B_i \rightsquigarrow E_i\}^{1\le i\le n}$. Assume that $\emptyset \vdash M : \Sigma \triangleright T \,/\, \square$. Let $\overline{v} = v_1, \cdots, v_n$ such that, for any $i \in [1,n]$, $v_i = \lambda x.\sigma_i(x; y.\,\mathsf{return}\, y)$. Then, $\mathbf{ET}([\![M]\!]^{\mathrm{e}}@(\overline{v^{\mathrm{h}}}, \underline{1})) = \mathbf{ET}(M)$.*

Corollary 1 (Decidability of Model Checking for $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$). *The higher-order model checking problem for $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ is decidable.*

6 Subtyping Extension

This section briefly sketches an extension $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ with subtyping. The subtyping judgments and rules, as well as the additional typing rules, are shown in Figure 9. The subtyping is similar to the one given by Kawamata et al. [24], generalizing (HT_Emb) to coerce $\square$ into an ATM effect everywhere and allowing width subtyping on operation signatures. The only subtlety is the side condition $A_1 \neq \square \Longrightarrow \square(\Sigma_1) <: \square(\Sigma_2)$ in the subtyping for computation types $\Sigma_1 \triangleright T_1 \,/\, A_1$ and $\Sigma_2 \triangleright T_2 \,/\, A_2$, which is highlighted by the gray box in Figure 9. We can prove type soundness of the extended $\mathsf{HEPCF}_{\square}^{\mathsf{ATM}}$ without this condition, but it is imposed for soundness of the CPS transformation. As suggested in the CPS type transformation (Figure 7), given a computation type $\Sigma \triangleright T \,/\, C^{\mathrm{ini}} \Rightarrow C^{\mathrm{fin}}$, the operation signature Σ occurs at a negative position (thus, the subtyping requires $\Sigma_2 <: \Sigma_1$), but $\square(\Sigma)$ occurs at a positive position. Thus, in coercing $\Sigma_1 \triangleright T_1 \,/\, A_1$ into $\Sigma_2 \triangleright T_2 \,/\, A_2$, if $A_1 \neq \square$ (which also ensures $A_2 \neq \square$), then $\square(\Sigma_1) <: \square(\Sigma_2)$ is required. It is left open whether we can provide a CPS transformation without this side condition or whether we can justify it without the lens of the CPS transformation.

The CPS transformation is also extended for subtyping. The extension transforms a subtyping derivation to a static function that coerces the subtype to the supertype. The definition is complicated but straightforward. The coercion functions are applied where typing derivations rely on subtyping.

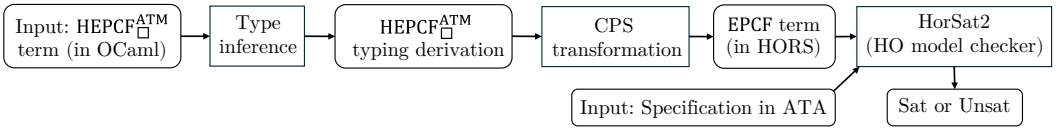


Fig. 10. Architecture of the implemented tool.

All the properties, such as type soundness and type and semantics preservation of the CPS transformation, are proven for the extension. See the supplementary material for the details including the full definition.

Now, we show how the program examples given in Section 2 are typechecked in this extension.

Example 6.1 (Nondeterministic Choice). The first example is $M_{\mathbf{D}}$, which uses the operation **Decide**. For that, consider the following term

$$\mathsf{let}\, f = \mathsf{return}\, \lambda x.\mathbf{Decide}(x; y.\, \mathsf{return}\, y)\, \mathsf{in}\, \mathsf{let}\, z = \mathsf{with}\, H\, \mathsf{handle}\, f\, ()\, \mathsf{in}\, \mathsf{case}(z; f\, (), \mathsf{return}\, \mathsf{false})\ ,$$

which is slightly modified from $M_{\mathbf{D}}$ to avoid ambiguity. Let $\Sigma_{\mathbf{D}} = \{\mathbf{Decide} : \mathsf{unit} \rightsquigarrow \mathsf{bool} \,/\, (\Sigma \triangleright 2 \,/\, \square) \Rightarrow (\Sigma \triangleright 2 \,/\, \square)\} \uplus \Sigma$ for some Σ, and assume that the effect handler H implements algebraic operations as specified by $\Sigma_{\mathbf{D}}$. Because $f : \mathsf{unit} \to \Sigma_{\mathbf{D}} \triangleright \mathsf{bool} \,/\, (\Sigma \triangleright 2 \,/\, \square) \Rightarrow (\Sigma \triangleright 2 \,/\, \square)$, the application $f\,()$ has the type $\Sigma_{\mathbf{D}} \triangleright \mathsf{bool} \,/\, (\Sigma \triangleright 2 \,/\, \square) \Rightarrow (\Sigma \triangleright 2 \,/\, \square)$. Therefore, the term with H handle $f\,()$ has the type $\Sigma \triangleright 2 \,/\, \square$, which can be coerced into $\Sigma_{\mathbf{D}} \triangleright 2 \,/\, (\Sigma \triangleright 2 \,/\, \square) \Rightarrow (\Sigma \triangleright 2 \,/\, \square)$. Since the case construct can have the same type as $f\,()$, the let construct let $z = \cdots$ can be given the type $\Sigma_{\mathbf{D}} \triangleright \mathsf{bool} \,/\, (\Sigma \triangleright 2 \,/\, \square) \Rightarrow (\Sigma \triangleright 2 \,/\, \square)$.

Here, it is crucial to instantiate the type $\Sigma \triangleright 2 \,/\, \square$ of the handling construct to the type $\Sigma_{\mathbf{D}} \triangleright 2 \,/\, (\Sigma \triangleright 2 \,/\, \square) \Rightarrow (\Sigma \triangleright 2 \,/\, \square)$ since the initial answer type of the handling construct must match with the final answer type $\Sigma \triangleright 2 \,/\, \square$ of the application $f\,()$. In fact, if the ATP effect $\square$ (or subtyping) were absent as in HEPCF$^{\mathsf{ATM}}$, the above term could not be typechecked. To see it, assume that the application $f\,()$ has a computation type $\Sigma_{\mathbf{D}} \triangleright \mathsf{bool} \,/\, C_1 \Rightarrow C_2$ for some initial answer type C_1 and final one C_2. The ATM type system assigns the type C_2 to the handling construct, and requires that the initial answer type of the type C_2 of the handling construct match with the final answer type C_2 of the application $f\,()$. However, without polymorphism (nor other typing mechanisms admitting circularity), this requirement would not be met because then C_2 cannot involve itself.

Example 6.2 (Exception Raising). Consider the recursive function $V_{\mathbf{R}}$, which is expressed as

$$\mathsf{fix}\, f.\lambda x.\mathsf{with}\, H\, \mathsf{handle}\, \mathbf{EOF}((); y.\, \mathsf{case}(y; \mathbf{Raise}((); z.\, \mathsf{return}\, z), \mathbf{Read}((); z.\, f\, ())))$$

in HEPCF$^{\mathsf{ATM}}_{\square}$. Let $\Sigma = \{\mathbf{EOF} : \mathsf{unit} \rightsquigarrow 2 \,/\, \square, \mathbf{Read} : \mathsf{unit} \rightsquigarrow \mathsf{unit} \,/\, \square\}$ and $\Sigma_{\mathbf{R}} = \{\mathbf{Raise} : \mathsf{unit} \rightsquigarrow \mathsf{unit} \,/\, (\Sigma \triangleright \mathsf{unit} \,/\, \square) \Rightarrow (\Sigma \triangleright \mathsf{unit} \,/\, \square)\} \uplus \Sigma$, and assume that the effect handler H implements algebraic operations as specified by $\Sigma_{\mathbf{R}}$. We show that the recursive function has the type $\mathsf{unit} \to \Sigma \triangleright \mathsf{unit} \,/\, \square$. This type assignment is possible if the handled term has the type $\Sigma_{\mathbf{R}} \triangleright \mathsf{unit} \,/\, (\Sigma \triangleright \mathsf{unit} \,/\, \square) \Rightarrow (\Sigma \triangleright \mathsf{unit} \,/\, \square)$ under the typing context $f : \mathsf{unit} \to \Sigma \triangleright \mathsf{unit} \,/\, \square, x : \mathsf{unit}$. Because the answer types of **EOF** and **Read** are polymorphic, the handled term has the same type as $\mathbf{Raise}((); z.\, \mathsf{return}\, z)$ and $f\,()$, which both can be of the desired type—the latter is achieved by subtyping $\Sigma \triangleright \mathsf{unit} \,/\, \square <: \Sigma_{\mathbf{R}} \triangleright \mathsf{unit} \,/\, (\Sigma \triangleright \mathsf{unit} \,/\, \square) \Rightarrow (\Sigma \triangleright \mathsf{unit} \,/\, \square)$.

Again, this example also requires polymorphism to be typechecked: although the return type of the recursive function would be required to involve itself as a final answer type, it would not be met without polymorphism (nor other recursive typing mechanisms).

7 Implementation

We extended the existing model checker `EffCaml` [56] for $\mathsf{HEPCF}^{\mathsf{ATM}}$ to implement our proof-of-concept model checker for $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$. The extended model checker, whose entire architecture is illustrated in Figure 10, verifies an $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$ program written in a subset of OCaml 5 against a safety property specified by an *alternating tree automaton* (ATA), which is an APTA with the parity condition that always holds (see the supplementary material for the definition of parity conditions). Our tool only handles ATAs, as our HOMC backend `HorSat2` [9, 30] only supports ATAs.

Given an $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$ term, our tool infers its type according to the type system of $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$. We implemented the type inference by following the approach of Kawamata et al. [24], as $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$ can be seen as a simplified version of their ATM refinement type system, with the exception of the inclusion of tail-resumptive operation clauses, which we identify by analyzing the term before the type inference. Our type inference is constraint-based: it first generates equality and subtyping constraints over value types, control effects, and operation signatures, and then solves the resulting constraints. Constraint generation and solving for operation signatures are handled in a way analogous to constraint-based record type inference, since subtyping on operation signatures resembles record subtyping. For control effects, we adopt the constraint-based type inference framework for the delimited control operators `shift0`/`reset0` [38]. We believe that our type inference implementation is sound, but this remains a conjecture.

If the type inference is successful—otherwise, it indicates that the input program is outside the fragment identified by $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$'s type system, so our tool aborts without passing the problem to the backend—then our tool transforms the $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$ term into a higher-order recursion scheme (HORS) in CPS, following the CPS transformation extended with subtyping (see Section 6). Finally, the output HORS is fed into `HorSat2`, which model checks the HORS against the given ATA.

Program	Specification	Sat/Unsat
$\mathbf{Open}\,();V_{\mathbf{R}}\,();\mathbf{Close}\,()$	File-Usage	Sat
$\mathbf{Open}\,();V_{\mathbf{R}}\,();\mathbf{Open}\,()$	File-Usage	Unsat
V true	No-Raise	Sat
V false	No-Raise	Unsat

We confirmed that our tool successfully verifies or falsifies four HOMC instances, as summarized in the right table. The first column presents the programs to be verified. In addition to the shorthand introduced in Section 5, we abbreviate $\sigma(V; x.\,\mathsf{return}\,x)$ to $\sigma\,V$ and use sequential composition $M_1; M_2$ and if branching $\mathsf{if}\,M\,\mathsf{then}\,M_1\,\mathsf{else}\,M_2$, which can easily be encoded in $\mathsf{HEPCF}^{\mathsf{ATM}}_{\square}$. The first two examples refer to $V_{\mathbf{R}}$, which is presented in Section 2 and expressed as

$$\mathsf{fix}\,f.\lambda x.\mathsf{with}\,H_{\mathbf{R}}\,\mathsf{handle}\,\mathsf{if}\,\mathbf{EOF}\,()\,\mathsf{then}\,\mathbf{Raise}\,()\,\mathsf{else}\,(\mathbf{Read}\,();f\,())$$

with $H_{\mathbf{R}} = \{\mathsf{return}\,x \mapsto \mathsf{return}\,x, \mathbf{EOF}(x) \mapsto \mathbf{EOF}\,(x), \mathbf{Read}(x) \mapsto \mathbf{Read}\,(x), \mathbf{Raise}(x;k) \mapsto \mathsf{return}\,x\}$, which forwards **EOF** and **Read** and handles the operation **Raise**. The functional value V in the third and fourth examples is defined to be

$$\mathsf{fix}\,f.\lambda x.\mathsf{let}\,g = (\mathsf{with}\,H_{\mathbf{S}}\,\mathsf{handle}\,\mathsf{let}\,y = \mathbf{Get}\,()\,\mathsf{in}\,\mathsf{if}\,y\,\mathsf{then}\,f\,y\,\mathsf{else}\,\mathbf{Raise}\,())\,\mathsf{in}\,g\,x$$

with $H_{\mathbf{S}} = \{\mathsf{return}\,x \mapsto \mathsf{return}\,\lambda y.\mathsf{return}\,x, \mathbf{Get}(x;k) \mapsto \mathsf{return}\,\lambda y.k\,y\,y, \mathbf{Raise}(x) \mapsto \mathbf{Raise}\,(x)\}$, which handles **Get** to return the current state y as in a state monad and forwards **Raise**. The argument x in V is an initial state of the handled term. The specifications File-Usage and No-Raise are both given as ATAs. The former describes that call sequences of **Open**, **Close**, **EOF**, and **Read** follow the regular expression $(\mathbf{Open}\,(\mathbf{EOF} \mid \mathbf{Read})^*\,\mathbf{Close})^*$ and **Read** is called only when it is immediately preceded by a call to **EOF** that returns true. The latter describes that no call to **Raise** escapes to the top level. See the artifact for the details of the specifications. All the verification tasks completed in less than 0.1 seconds on the machine with 12th Gen Intel(R) Core(TM) i7-1270P 2.20 GHz, 32 GB of memory. This result demonstrates that the proposed CPS transformation enables

model checking of the fragment identified by our type system. That said, evaluating the performance of our tool on larger and more complex instances remains future work.

8 Related Work

Higher-Order Model Checking. Model checking of higher-order programs has been an active research topic in the last twenty-five years, giving rise to many positive, but also negative, results. We should certainly mention the pioneering and partial results by Knapik et al. [25, 26], Ong's breakthrough result [42] about the decidability of the HOMC problem for higher-order recursion schemes, and Kobayashi and co-authors' work on model checking as (intersection) type checking [33]. Some studies do exist about extensions of the cited decidability results to calculi endowed with some *specific* form of effects [27, 37, 51], but all this has been given a clearer status by Dal Lago and Ghyselen [14], who recently studied the problem of HOMC for functional languages with algebraic effects, giving a decidability result for the HOMC problem holding when specifications are expressed in an APTA. This is not inconsistent with the aforementioned undecidability results about, e.g., probabilistic choice [32], as the specifications one is interested at there *cannot* be formalized as APTAs. The recent paper by Kobayashi [31] further helps in understanding where the source of undecidability actually lies, and why linearity is the key to design decidable fragments.

Effectful Higher-Order Programs and Handlers. Since Moggi's seminal work on monads [39], the theory of languages with both higher-order functions and effects has been structured around the categorical notion of monads. All this has been given a more operational flavor by Plotkin and Power in their work about algebraic effects [46]. Algebraic operations can be given a computational meaning through effect handlers, this way allowing effects to be interpreted by the program itself rather than by the environment, in the style of the `try – with` operator for exceptions. The theory and practice of effect handling has been extensively studied [5, 21–23, 47, 48, 53, 54, 66]. Dal Lago and Ghyselen [14] show that if the underlying calculus is along the lines of the aforementioned ones, the HOMC problem becomes undecidable even for a simply-typed discipline. Sekiyama and Unno [56] have recently showed that in the presence of answer-type modifications decidability can be recovered, but that to do so one must renounce to any form of polymorphism, and have algebraic operations typed in a monomorphic way. We show that Sekiyama and Unno's approach is extensible to two forms of polymorphism: answer-type polymorphism and subtyping.

Other Approaches to Temporal Verification of Effectful Higher-Order Programs. A recent line of work has been concerned with the temporal verification of infinite-state higher-order effectful programs using *type-and-effect systems*. Gordon [20] defines a framework for sequential effects with tagged control operators akin to `abort` and `call/cc`, capturing temporal safety properties. Similarly, Sekiyama and Unno [55] give a type-and-effect system for general temporal properties in the presence of the control operators `shift0` and `reset0`. Song et al. [60] tackles the safety verification problem for general effect handlers against specifications in a logic more expressive than classical LTL. We are not aware of any work dealing with the problem of verifying general effect handlers against APTAs except for that of Dal Lago and Ghyselen [14] and that of Sekiyama and Unno [56]. For temporal verification on primitive effects or trace properties, numerous frameworks have been proposed [19, 40, 55, 58, 59, 67, 69], but extensions to effect handlers or other control operators may require nontrivial efforts because their ability to manipulate continuations could bring unexpected issues, as observed by Dal Lago and Ghyselen [14] or de Vilhena and Pottier [16].

Answer-Type Polymorphism and CPS Transformation. Answer-type polymorphism has been discovered by Riecke and Thielecke [50] as a proof technique, and later Thielecke [61] linked answer-type polymorphism to effect systems—the answer types of terms guaranteed to be pure by

a type system can be polymorphic. Answer-type polymorphism has been introduced to ATM type systems by Asai and Kameyama [1] and Materzok and Biernacki [38]. While Asai and Kameyama's type system allows universally quantifying answer types, Materzok and Biernacki do not introduce parametric polymorphism but instead offers the type constructor ϵ to represent answer-type polymorphism (corresponding to $\square$ in our notation), as well as subtyping for instantiating ϵ. Materzok and Biernacki also defined a selective type-directed CPS transformation that can transform terms of the answer type ϵ to terms that do not rely on continuations. However, they did not show the semantic preservation of the CPS transformation; it is only shown that *weak* semantic preservation—i.e., source terms before and after reduction can be transformed into $\beta\eta$-equivalent terms—for *untyped* source terms using a *non-selective*, *non-type-directed* CPS transformation. Kawamata et al. [24] brought answer-type polymorphism to effect handlers and gave a type-directed CPS transformation. Their CPS transformation satisfies *strong* semantic preservation—i.e., it preserves reduction—but it relies on parametric polymorhpism in the target language as in the one of Hillerström et al. [22]. Also, their type system did not support ATP algebraic operations. Asai and Uehara [2] formalized a selective, type-directed CPS transformation in the presence of answer-type polymorphism (in the style of Asai and Kameyama [1]) and proved that it satisfies a weak semantic preservation, but it is not proven to preserve observational behavior. Sato et al. [51] proposed a selective CPS transformation for HOMC and showed that it satisfies strong semantic preservation, but they address neither control operators (at least formally) nor answer-type polymorhpism. Our selective, type-directed CPS transformation for effect handlers in the presence of answer-type polymorphism does not satisfy strong semantic preservation, but it preserves reduction modulo full $\beta\eta$ monadic reduction, which is enough to guarantee the preservation of observational behavior.

9 Conclusion

We showed that the HOMC problem for effect handlers remains decidable in the presence of answer-type polymorphism and subtyping by giving a selective CPS transformation that turns answer-type-polymorphic terms to continuation-independent terms. There are several future directions. Because effect handlers can be viewed as transducers on computation trees [49], it is an interesting question whether the verification technique based on *higher-order tree transducers* could be applied to effect handlers. To make the verification problem decidable, Kobayashi et al. [34] assume the *linearity* on trees to be verified, i.e., trees to be verified are traversed only once. To verify non-linear trees, they require *coercion annotations*. Because the ATM-based approaches allow traversing computation trees multiple times without annotations, we first need to explore the root cause of the gap. The effect handlers that this paper focused on are called *dynamically scoped* and *deep* [23], but there are many other forms of effect handlers, such as lexically scoped [6, 8], shallow [21], scoped [64], and bidirectional [68] effect handlers. Kobayashi [31] studied the HOMC problem in the presence of other effectful features such as local store. One of the long-term goals is to make a unified model checking framework to accommodate these various effectful features.

Data Availability Statements

The artifact [52] provides the supplementary material and a document for the implemented HOMC tool and the reproduction of the experimental results in Section 7. The tool, as well as the benchmarks, is also found at https://github.com/hiroshi-unno/coar.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was partly supported by JSPS KAKENHI (JP20H04162, JP20H05703, JP22H03564, JP22H03570, JP24H00699, JP25H00446), JST CREST (JPMJCR21M3), and the ANR Project HOPR (ANR-24-CE48-5521-01).

References

[1] Kenichi Asai and Yukiyoshi Kameyama. 2007. Polymorphic Delimited Continuations. In *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Proceedings*. 239–254. doi:10.1007/978-3-540-76637-7_16

[2] Kenichi Asai and Chihiro Uehara. 2018. Selective CPS transformation for shift and reset. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, Los Angeles, CA, USA, January 8-9, 2018*, Fritz Henglein and Hsiang-Shang Ko (Eds.). ACM, 40–52. doi:10.1145/3162069

[3] David Basin, Cas Cremers, and Catherine Meadows. 2018. *Model Checking Security Protocols*. Springer International Publishing, Cham, 727–762. doi:10.1007/978-3-319-10575-8_22

[4] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123. doi:10.1016/j.jlamp.2014.02.001

[5] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting algebraic effects. *Proc. ACM Program. Lang.* 3, POPL (2019), 6:1–6:28. doi:10.1145/3290319

[6] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *PACMPL* 4, POPL (2020), 48:1–48:29. doi:10.1145/3371116

[7] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2018. Effect handlers for the masses. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 111:1–111:27. doi:10.1145/3276481

[8] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 126:1–126:30. doi:10.1145/3428194

[9] Christopher H. Broadbent and Naoki Kobayashi. 2013. Saturation-Based Model Checking of Higher-Order Recursion Schemes. In *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy (LIPIcs, Vol. 23)*, Simona Ronchi Della Rocca (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 129–148. doi:10.4230/LIPICS.CSL.2013.129

[10] Edmund M. Clarke and E. Allen Emerson. 1981. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981 (Lecture Notes in Computer Science, Vol. 131)*, Dexter Kozen (Ed.). Springer, 52–71. doi:10.1007/BFB0025774

[11] Edmund M. Clarke, Anubhav Gupta, Himanshu Jain, and Helmut Veith. 2005. Model Checking: Back and Forth between Hardware and Software. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions (Lecture Notes in Computer Science, Vol. 4171)*, Bertrand Meyer and Jim Woodcock (Eds.). Springer, 251–255. doi:10.1007/978-3-540-69149-5_27

[12] Youyou Cong and Kenichi Asai. 2022. Understanding Algebraic Effect Handlers via Delimited Control Operators. In *Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 13401)*, Wouter Swierstra and Nicolas Wu (Eds.). Springer, 59–79. doi:10.1007/978-3-031-21314-4_4

[13] Ugo Dal Lago, Francesco Gavazzo, and Paul Blain Levy. 2017. Effectful applicative bisimilarity: Monads, relators, and Howe's method. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 1–12. doi:10.1109/LICS.2017.8005117

[14] Ugo Dal Lago and Alexis Ghyselen. 2024. On Model-Checking Higher-Order Effectful Programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 2610–2638. doi:10.1145/3632929

[15] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming*. 151–160. doi:10.1145/91556.91622

[16] Paulo Emílio de Vilhena and François Pottier. 2021. A separation logic for effect handlers. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. doi:10.1145/3434314

[17] Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Sci. Comput. Program.* 17, 1-3 (1991), 35–75. doi:10.1016/0167-6423(91)90036-W

[18] Dan R. Ghica, Sam Lindley, Marcos Maroñas Bravo, and Maciej Piróg. 2022. High-level effect handlers in C++. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1639–1667. doi:10.1145/3563445

[19] Colin S. Gordon. 2017. A Generic Approach to Flow-Sensitive Polymorphic Effects. In *31st European Conference on Object-Oriented Programming, ECOOP 2017 (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:31. doi:10.4230/LIPIcs.ECOOP.2017.13

[20] Colin S. Gordon. 2020. Lifting Sequential Effects to Control Operators. In *34th European Conference on Object-Oriented Programming, ECOOP 2020 (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:30. doi:10.4230/LIPIcs.ECOOP.2020.23

[21] Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 415–435. doi:10.1007/978-3-030-02768-1_22

[22] Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017*. 18:1–18:19. doi:10.4230/LIPIcs.FSCD.2017.18

[23] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP 2013*. 145–158. doi:10.1145/2500365.2500590

[24] Fuga Kawamata, Hiroshi Unno, Taro Sekiyama, and Tachio Terauchi. 2024. Answer Refinement Modification: Refinement Type System for Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 8, POPL (2024), 115–147. doi:10.1145/3633280

[25] Teodor Knapik, Damian Niwinski, and Pawel Urzyczyn. 2001. Deciding Monadic Theories of Hyperalgebraic Trees. In *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2044)*, Samson Abramsky (Ed.). Springer, 253–267. doi:10.1007/3-540-45413-6_21

[26] Teodor Knapik, Damian Niwinski, and Pawel Urzyczyn. 2002. Higher-Order Pushdown Trees Are Easy. In *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2303)*, Mogens Nielsen and Uffe Engberg (Eds.). Springer, 205–222. doi:10.1007/3-540-45931-6_15

[27] Naoki Kobayashi. 2009. Model-checking higher-order functions. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, António Porto and Francisco Javier López-Fraguas (Eds.). ACM, 25–36. doi:10.1145/1599410.1599415

[28] Naoki Kobayashi. 2009. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 416–428. doi:10.1145/1480881.1480933

[29] Naoki Kobayashi. 2013. Model Checking Higher-Order Programs. *J. ACM* 60, 3 (2013), 20:1–20:62. doi:10.1145/2487241.2487246

[30] Naoki Kobayashi. 2016. HorSat2: A Saturation-Based Model Checker for Higher-Order Recursion Schemes. Private communication. Available at https://github.com/hopv/horsat2..

[31] Naoki Kobayashi. 2025. On Decidable and Undecidable Extensions of Simply Typed Lambda Calculus. *Proc. ACM Program. Lang.* 9, POPL (2025), 1136–1166. doi:10.1145/3704875

[32] Naoki Kobayashi, Ugo Dal Lago, and Charles Grellois. 2019. On the Termination Problem for Probabilistic Higher-Order Recursive Programs. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*. IEEE, 1–14. doi:10.1109/LICS.2019.8785679

[33] Naoki Kobayashi and C.-H. Luke Ong. 2009. A Type System Equivalent to the Modal Mu-Calculus Model Checking of Higher-Order Recursion Schemes. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009*. IEEE Computer Society, 179–188. doi:10.1109/LICS.2009.29

[34] Naoki Kobayashi, Naoshi Tabuchi, and Hiroshi Unno. 2010. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 495–508. doi:10.1145/1706299.1706355

[35] Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*. 486–499. http://dl.acm.org/citation.cfm?id=3009872

[36] Paul Blain Levy. 2001. *Call-by-push-value*. Ph. D. Dissertation. Queen Mary University of London, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.369233

[37] Mark Lillibridge. 1999. Unchecked Exceptions Can Be Strictly More Powerful Than Call/CC. *High. Order Symb. Comput.* 12, 1 (1999), 75–104. doi:10.1023/A:1010020917337

[38] Marek Materzok and Dariusz Biernacki. 2011. Subtyping Delimited Continuations. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 81–93. doi:10.1145/2034773.2034786

[39] Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. 14–23. doi:10.1109/LICS.1989.39155

[40] Yoji Nanjo, Hiroshi Unno, Eric Koskinen, and Tachio Terauchi. 2018. A Fixpoint Logic and Dependent Effects for Temporal Property Verification. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*, Anuj Dawar and Erich Grädel (Eds.). ACM, 759–768. doi:10.1145/3209108.3209204

[41] Lasse R. Nielsen. 2001. A Selective CPS Transformation. In *Seventeenth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2001, Aarhus, Denmark, May 23-26, 2001 (Electronic Notes in Theoretical Computer Science, Vol. 45)*, Stephen D. Brookes and Michael W. Mislove (Eds.). Elsevier, 311–331. doi:10.1016/S1571-0661(04)80969-1

[42] C.-H. Luke Ong. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 81–90. doi:10.1109/LICS.2006.38

[43] Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, K. C. Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 460–485. doi:10.1145/3622814

[44] Gordon D. Plotkin. 1977. LCF Considered as a Programming Language. *Theor. Comput. Sci.* 5, 3 (1977), 223–255. doi:10.1016/0304-3975(77)90044-5

[45] Gordon D. Plotkin and A. John Power. 2002. Computational Effects and Operations: An Overview. In *Proceedings of the Workshop on Domains VI 2002, Birmingham, UK, September 16-19, 2002 (Electronic Notes in Theoretical Computer Science, Vol. 73)*, Martín Escardó and Achim Jung (Eds.). Elsevier, 149–163. doi:10.1016/J.ENTCS.2004.08.008

[46] Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. doi:10.1023/A:1023064908962

[47] Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, Proceedings*. 80–94. doi:10.1007/978-3-642-00590-9_7

[48] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013). doi:10.2168/LMCS-9(4:23)2013

[49] Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. In *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015 (Electronic Notes in Theoretical Computer Science, Vol. 319)*, Dan R. Ghica (Ed.). Elsevier, 19–35. doi:10.1016/J.ENTCS.2015.12.003

[50] Jon G. Riecke and Hayo Thielecke. 1999. Typed Exeptions and Continuations Cannot Macro-Express Each Other. In *Automata, Languages and Programming, 26th International Colloquium, ICALP'99, Prague, Czech Republic, July 11-15, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1644)*, Jirí Wiedermann, Peter van Emde Boas, and Mogens Nielsen (Eds.). Springer, 635–644. doi:10.1007/3-540-48523-6_60

[51] Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. 2013. Towards a scalable software model checker for higher-order programs. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013*, Elvira Albert and Shin-Cheng Mu (Eds.). ACM, 53–62. doi:10.1145/2426890.2426900

[52] Taro Sekiyama, Ugo Dal Lago, and Hiroshi Unno. 2025. *Artifact for "On Higher-Order Model Checking of Effectful Answer-Type-Polymorphic Programs"*. doi:10.5281/zenodo.16923662

[53] Taro Sekiyama and Atsushi Igarashi. 2019. Handling Polymorphic Algebraic Effects. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Proceedings*. 353–380. doi:10.1007/978-3-030-17184-1_13

[54] Taro Sekiyama, Takeshi Tsukada, and Atsushi Igarashi. 2020. Signature restriction for polymorphic algebraic effects. *Proc. ACM Program. Lang.* 4, ICFP (2020), 117:1–117:30. doi:10.1145/3408999

[55] Taro Sekiyama and Hiroshi Unno. 2023. Temporal Verification with Answer-Effect Modification: Dependent Temporal Type-and-Effect System with Delimited Continuations. *Proc. ACM Program. Lang.* 7, POPL, Article 71 (2023), 32 pages. doi:10.1145/3571264

[56] Taro Sekiyama and Hiroshi Unno. 2024. Higher-Order Model Checking of Effect-Handling Programs with Answer-Type Modification. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 2662–2691. doi:10.1145/3689805

[57] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 206–221. doi:10.1145/3453483.3454039

[58] Christian Skalka and Scott F. Smith. 2004. History Effects and Verification. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004 (Lecture Notes in Computer Science, Vol. 3302)*, Wei-Ngan Chin (Ed.). Springer, 107–128. doi:10.1007/978-3-540-30477-7_8

[59] Christian Skalka, Scott F. Smith, and David Van Horn. 2008. Types and trace effects of higher order programs. *J. Funct. Program.* 18, 2 (2008), 179–249. doi:10.1017/S0956796807006466

[60] Yahui Song, Darius Foo, and Wei-Ngan Chin. 2022. Automated Temporal Verification for Algebraic Effects. In *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022 (Lecture Notes in Computer Science, Vol. 13658)*, Ilya Sergey (Ed.). Springer, 88–109. doi:10.1007/978-3-031-21037-2_5

[61] Hayo Thielecke. 2003. From control effects to typed continuation passing. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 139–149. doi:10.1145/640128.604144

[62] Takeshi Tsukada and Naoki Kobayashi. 2010. Untyped Recursion Schemes and Infinite Intersection Types. In *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6014)*, C.-H. Luke Ong (Ed.). Springer, 343–357. doi:10.1007/978-3-642-12032-9_24

[63] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. doi:10.1006/inco.1994.1093

[64] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Haskell 2014*. 1–12. doi:10.1145/2633357.2633358

[65] Ningning Xie and Daan Leijen. 2020. Effect handlers in Haskell, evidently. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2020, Virtual Event, USA, August 7, 2020*, Tom Schrijvers (Ed.). ACM, 95–108. doi:10.1145/3406088.3409022

[66] Takuma Yoshioka, Taro Sekiyama, and Atsushi Igarashi. 2024. Abstracting Effect Systems for Algebraic Effect Handlers. *Proc. ACM Program. Lang.* 8, ICFP (2024), 455–484. doi:10.1145/3674641

[67] Yongwei Yuan, Zhe Zhou, Julia Belyakova, and Suresh Jagannathan. 2025. Derivative-Guided Symbolic Execution. *Proc. ACM Program. Lang.* 9, POPL (2025), 1475–1505. doi:10.1145/3704886

[68] Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. 2020. Handling bidirectional control flow. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 139:1–139:30. doi:10.1145/3428207

[69] Zhe Zhou, Qianchuan Ye, Benjamin Delaware, and Suresh Jagannathan. 2024. A HAT Trick: Automatically Verifying Representation Invariants using Symbolic Finite Automata. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1387–1411. doi:10.1145/3656433

Received 2025-03-26; accepted 2025-08-12