Higher-Order Model Checking of Effect-Handling Programs with Answer-Type Modification

TARO SEKIYAMA, National Institute of Informatics, Japan and SOKENDAI, Japan
HIROSHI UNNO, Tohoku University, Japan

Since the seminal work by Ong, the model checking of higher-order programs—called *higher-order model checking*, or HOMC for short—has gained attention. It is also crucial for making HOMC applicable to real-world software to address programs involving computational effects. Recently, Dal Lago and Ghyselen considered an extension of HOMC to *algebraic effect handlers*, which enable programming the semantics of effects. They showed a negative result for HOMC with algebraic effect handlers—it is undecidable. In this work, we explore a restriction on programs with algebraic effect handlers which ensures the decidability of HOMC while allowing implementations of various effects. We identify the crux of the undecidability as the use of an unbounded number of algebraic effect handlers being active at the same time. To prevent it, we introduce *answer-type modification* (ATM), which can bound the number of algebraic effect handlers that can be active at the same time. We prove that ATM can ensure the decidability of HOMC and show that it accommodates a wide range of effects. To evaluate our approach, we implemented an automated verifier EffCaml based on the presented techniques and confirmed that the program examples discussed in this paper can be automatically verified.

CCS Concepts: • **Theory of computation** → **Type theory**; **Verification by model checking**.

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

ACM Reference Format:
Taro Sekiyama and Hiroshi Unno. 2024. Higher-Order Model Checking of Effect-Handling Programs with Answer-Type Modification. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 365 (October 2024), 30 pages. https://doi.org/10.1145/3689805

1 Introduction

1.1 Background: Higher-Order Model Checking of Effectful Programs

Model checking [Baier and Katoen 2008; Clarke et al. 2018] is one of the successful program verification methodologies, representing the program to be verified as a structure and the desired property as a logical formula and checking if the formula holds under the structure, that is, if the structure is a *model* of the formula. Thus far, model checking has been extensively studied from various perspectives, including efficient algorithms [Bradley 2011; Burch et al. 1990], logics to specify desired properties [Pnueli 1977], and programs to be verified [Jhala and Majumdar 2009].

Higher-order model checking (HOMC) [Kobayashi 2013] aims to model check higher-order programs. A breakthrough in the computability of HOMC was achieved by Ong [2006], who showed that the model checking of *higher-order recursion schemes* (HORS), tree grammars to generate infinite trees using higher-order functions, for formulas in monadic second-order logic (MSO) is decidable. Since Ong's seminal work, HOMC has gained attention in its theoretical aspects [Hague et al. 2008; Kobayashi and Ong 2009a; Salvati and Walukiewicz 2014], applied to various verification

Authors' Contact Information: Taro Sekiyama, National Institute of Informatics, Tokyo, Japan and SOKENDAI, Tokyo, Japan, tsekiyama@acm.org; Hiroshi Unno, Tohoku University, Sendai, Japan, hiroshi.unno@acm.org.

ACM 2475-1421/2024/10-ART365
https://doi.org/10.1145/3689805

problems [Kobayashi and Igarashi 2013] and programming constructs [Kobayashi et al. 2010; Ong and Ramsay 2011], and led to the development of automated verification tools [Kobayashi et al. 2011, 2010; Ramsay et al. 2014]. Although HOMC with infinite data domains (such as natural numbers) is undecidable in general [Kobayashi et al. 2010], there are several studies to apply HOMC to infinite data domains at the cost of giving up completeness or decidability [Kobayashi and Igarashi 2013; Kobayashi et al. 2011, 2010; Matsumoto et al. 2015; Ong and Ramsay 2011; Unno et al. 2010].

Another crucial research line for making HOMC applicable to real-world software is to address programs involving *computational effects*. Kobayashi [2009] showed that the behavior of nondeterministic branches can be naturally encoded into HORS. Sato et al. [2013] demonstrated that the HOMC problem for programs with exception handling or control operator call/cc [Reynolds 1972] can be reduced to the one for programs with none of them by transformation into continuation passing style (CPS). Kobayashi et al. [2019] introduced a probabilistic extension of HORS for model checking higher-order probabilistic programs, although the almost sure termination problem for the extended HORS is undecidable. In spite of these attempts in the literature, it had been unclear whether HOMC can be extended to general effects such as mutable state, input/output, and concurrency, as well as the effects mentioned above, and, if it is possible, whether the HOMC problem for general effects is decidable (under some appropriate logics to describe properties).

Dal Lago and Ghyselen [2024] tackled this challenge by considering two effectful extensions of finitary PCF (i.e., PCF [Plotkin 1977] only with finite data domains). The first extension EPCF is equipped with *algebraic effects* [Plotkin and Power 2003].[1] Dal Lago and Ghyselen equipped EPCF with a computation tree semantic where effect operations work as tree constructors and showed that the model checking problem for trees generated by EPCF programs against an MSO formula is decidable. The second extension HEPCF supports *algebraic effect handlers* (effect handlers, for short) [Plotkin and Pretnar 2009, 2013], which allow the users to implement their own effects using delimited continuations. In the computation tree semantics of HEPCF, only unhandled effect operations appear in a generated tree. Thus, in HEPCF, we can see handled effects as user-defined effects and unhandled effects as primitive (or built-in) effects, and the behavior of the primitive effects can be specified by logical formulas that exclude the behavior unintended as the primitive effects of interest. A critical difference between EPCF and HEPCF is that the HOMC problem for HEPCF programs against MSO formulas is *undecidable*. To show this negative result, Dal Lago and Ghyselen provided an encoding of natural numbers in HEPCF, where an arbitrary natural number n is represented by n calls to an effect operation and the deconstruct operation on natural numbers are implemented using an effect handler. Because the existence of an infinite data domain makes the HOMC problem undecidable, the HOMC problem for HEPCF turns out to be undecidable. On a positive side, Dal Lago and Ghyselen also showed that the HOMC problem for a restricted class of effect handlers is decidable; PCF allowing only such effect handlers are called Generic Effects PCF (GEPCF for short), as the restricted effect handlers are expressive enough to implement *generic effects* [Plotkin and Power 2003]. However, GEPCF is too restrictive to allow implementing broadly used effects such as nondeterminism and exceptions. Because the HOMC problem with nondeterminism or exception handling is decidable [Kobayashi 2009; Sato et al. 2013], there should be a more permissive class of effect handlers for which the HOMC problem remains decidable.

1.2 This Work

We aim to establish an expressive class of effect handlers that allow implementing a wide range of effects and preserve the decidability of HOMC. Specifically, we define a type system to identify

[1] More precisely, EPCF only supports effect operations and their semantics are formulated using a logical specification, while the semantics of algebraic effects are given by an equational theory in the original work [Plotkin and Power 2003].

such a class of effect handlers. Our type system is based on the one given by Kawamata et al. [2024], who adapted the notion of *answer-type modification* (ATM) [Cong and Asai 2022; Cong et al. 2021; Danvy and Filinski 1990; Ishio and Asai 2022; Kameyama and Yonezawa 2008; Materzok and Biernacki 2011; Sekiyama and Unno 2023] to effect handlers for precise tracking of value flows. A crucial feature of ATM for our aim is that *ATM can also bound the number of effect handlers that can be active (i.e., ready to handle effects) at the same time*. When HEPCF is endowed with our type system—which we call HEPCF$^{\mathsf{ATM}}$—encoded natural numbers can be decomposed only a number of times which is bounded statically. Thus, it prevents the full encoding of the infinite data domain, making the HOMC problem decidable. To formally prove it, we define a type- and semantics-preserving CPS transformation from HEPCF$^{\mathsf{ATM}}$ to EPCF. With this transformation, the computability of HOMC for HEPCF$^{\mathsf{ATM}}$ can be reduced to that for EPCF.

The contributions of the present work are summarized as follows:

- We define HEPCF$^{\mathsf{ATM}}$, a λ-calculus equipped with algebraic effects and handlers and a type system allowing ATM. While HEPCF of Dal Lago and Ghyselen restricts the parameter and arity type of every effect operation to be a finite base and enumerate type, respectively, HEPCF$^{\mathsf{ATM}}$ does not impose such a restriction for handled effects and allows their operations to take and return higher-order values.
- We define a CPS transformation from HEPCF$^{\mathsf{ATM}}$ to EPCF and show that it preserves typing and semantics of given HEPCF$^{\mathsf{ATM}}$ programs. As Plotkin's colon translation [Plotkin 1975], our CPS transformation produces only EPCF terms that involve no *administrative reduction*, which is an "extra" reduction in that there is no corresponding reduction in the original HEPCF$^{\mathsf{ATM}}$ terms. The guarantee of no administrative reduction enables us to show a precise semantic relationship between HEPCF$^{\mathsf{ATM}}$ terms and their CPS-transformation results.
- To evaluate our approach, we implemented an automated verifier EffCaml for HEPCF$^{\mathsf{ATM}}$ programs based on the presented techniques: we implemented type inference for HEPCF$^{\mathsf{ATM}}$ and the CPS transformation from HEPCF$^{\mathsf{ATM}}$ to EPCF, and integrated them with the higher-order model checker HorSat2 [Broadbent and Kobayashi 2013; Kobayashi 2016]. We also confirmed that our tool can automatically verify the examples discussed in the paper.

The rest of the paper is organized as follows. Section 2 gives an overview of the present work. It starts by reviewing the work of Dal Lago and Ghyselen [2024] and why the HOMC problem for HEPCF is undecidable and then explains our idea to prevent it. We define EPCF and its HOMC problem in Section 3 for making the paper self-contained. Section 4 introduces HEPCF$^{\mathsf{ATM}}$. Section 5 defines the HOMC problem for HEPCF$^{\mathsf{ATM}}$, presents the CPS transformation from HEPCF$^{\mathsf{ATM}}$ to EPCF, and shows the decidability of the HOMC for HEPCF$^{\mathsf{ATM}}$ via the CPS transformation. Section 6 describes the implemented verifier EffCaml. Finally, we discuss the related work in more detail in Section 7 and conclude in Section 8. In the paper, we only state key lemmas and theorems. The other auxiliary lemmas and the detailed proofs of all the lemmas and theorems are found in the supplementary material. Our verifier EffCaml is available at https://github.com/hiroshi-unno/coar.

2 Overview

This section reviews in detail how Dal Lago and Ghyselen [2024] address HOMC for algebraic effects (Section 2.1) and HOMC for effect handlers, as well as why the latter is undecidable (Section 2.2). We then explain why ATM can ensure the decidability of HOMC for effect handlers (Section 2.3).

2.1 Model Checking of Effectful Programs

In *algebraic effects* [Plotkin and Power 2003], effectful behavior arises by *algebraic operations*. An algebraic operation is a producer of an effect, which influences the behavior of the remaining

```
let x = ref true
let rec f g =
  let b1 = !x in
  if g b1 then ()
  else
    let b2 = !x in
    if b1 = b2 then f g
    else raise ()
...
```

(a) In ML-like syntax.

$$
\begin{array}{l}
\mathbf{Set}(\mathsf{true}; \_. \\
\mathsf{let}\ f = \mathsf{fix}\, f.\, \lambda g. \\
\quad \mathbf{Get}((); b_1. \\
\quad \mathsf{if}\ g\, b_1\ \mathsf{then}\ () \\
\quad \mathsf{else} \\
\quad\quad \mathbf{Get}((); b_2. \\
\quad\quad \mathsf{if}\ b_1 = b_2\ \mathsf{then}\ f\, g \\
\quad\quad \mathsf{else}\ \mathbf{Raise}((); x.\, \mathsf{absurd}\, x))) \\
\mathsf{in}\ \cdots
\end{array}
$$

(b) In EPCF.

Fig. 1. A program example with state manipulation and exception raising.

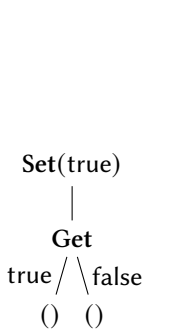
computation, that is, the *continuation*. Thus, algebraic effects are incorporated to the λ-calculus by adding constructs of the form $\sigma(v; x.\, e)$, which calls the algebraic operation σ with parameter v and carries its continuation $x.\, e$. Dal Lago and Ghyselen [2024] define a λ-calculus EPCF with algebraic effects in such a way.

For instance, consider the higher-order recursive function `f` in Figure 1a. In this example, variable `x` is a global Boolean reference and function `raise` raises an exception that signals a run-time error. The function `f` raises an exception if an argument function `g` returns false for the Boolean value `b1` that the reference `x` possesses at the time when `f` is called, and if the application of `g` changes the value pointed to by `x`; otherwise, the computation finishes or `f` is called recursively. Thus, the application `f (fun b -> true)` immediately terminates, `f (fun b -> false)` diverges, and `f (fun b -> x := not b; false)` ends with a run-time error.

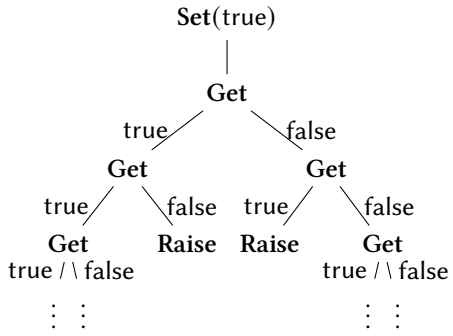
Consider expressing this example using algebraic effects. We assume the operations **Set** for the assignment to `x`, **Get** for the dereference of `x`, and **Raise** for the exception raising. The type interface of algebraic operations is called an *operation signature*. For the above operations, we can assume signature $\Xi \stackrel{\text{def}}{=} \{\, \mathbf{Set} : \mathsf{bool} \rightsquigarrow \mathsf{unit},\ \mathbf{Get} : \mathsf{unit} \rightsquigarrow \mathsf{bool},\ \mathbf{Raise} : \mathsf{unit} \rightsquigarrow \mathsf{empty} \,\}$ (empty is the type having no inhabitant).[2] We also assume the fixpoint operation $\mathsf{fix}\, x.v$ to encode recursive functions. Then, the example can be rewritten as in Figure 1b, which is similar to the program in Figure 1a except that the algebraic operations in Figure 1b take the continuations as arguments (absurd is a construct that can be given an arbitrary type if a value of the type `empty` is passed).

To interpret an EPCF term as a structure to be model checked, Dal Lago and Ghyselen [2024] equip EPCF with a computation tree semantics [Lindley 2014; Pretnar 2015], under which the EPCF term is regarded to generate a possibly infinite tree. For instance, Figure 2 shows the computation trees generated by the terms corresponding to `f (fun b -> true)`, `f (fun b -> false)`, or `f (fun b -> x := not b; false)`. Each internal node of a computation tree is an algebraic operation and its subtrees are generated by the continuation of the operation with arguments labeled to the corresponding edges. Leaf nodes denote the final evaluation results. Because any path of the tree in Figure 2a is finite, it is found that `f (fun b -> true)` terminates. As for the tree in Figure 2b, we can reach a **Raise** node if and only if consecutive calls to **Get** return different Boolean values. Because it is inconsistent with the behavior of mutable references, we can conclude that no exception will be raised under the usual semantics of mutable state. By taking a look at the tree in Figure 2c, we can find that an exception will be raised because a **Raise** node is reached if **Get** returns a Boolean b

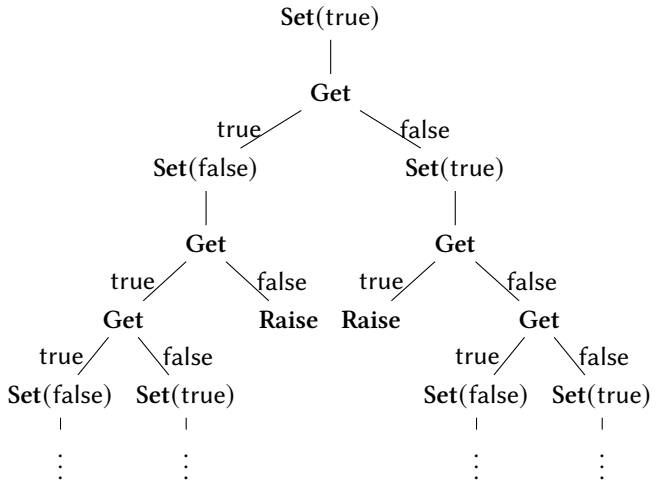
[2]Here **Set** and **Get** are supposed to be for the specific reference `x`, but we can easily address multiple global references by, e.g., preparing distinct operations for each reference or parameterizing the operations over the locations of references.



(a) $f\ (\lambda b.\,\mathsf{true})$.



(b) $f\ (\lambda b.\,\mathsf{false})$.



(c) $f\ (\lambda b.\,\mathsf{let}\ b' = \mathsf{not}\ b\ \mathsf{in}\ \mathbf{Set}(b'); \_.\,\mathsf{false}))$.

Fig. 2. Computation trees of λ-terms with algebraic effects. The unit parameter and continuation unable to be executed are omitted.

after **Set**(b) is invoked, which is the standard behavior of mutable state. We can specify the desired property—no exception will be raised—as well as the expected behavior of the effects—e.g., **Get** returns a Boolean b if the parameter of **Set** called immediately before it is b, and consecutive calls to **Get** return the same value—as an MSO formula, or equivalently, as an *alternating parity tree automaton* (APT for short) [Kobayashi and Ong 2009b; Ong 2006]. Dal Lago and Ghyselen [2024] considered the problem of whether the computation tree generated by an EPCF term is accepted by an APT and showed that it can be solved in a decidable manner (provided that all the data domains in EPCF are finite). We will introduce the formal definition of APTs and show certain instances of the HOMC problem for EPCF in Section 3.2.

2.2 Model Checking of Effect-Handling Programs

In the semantics presented in the previous section, the meaning of algebraic effects is fixed during evaluating programs since they are formulated via an APT. This is natural when algebraic effects model "primitive" effects implemented by the run-time system of the language of interest.

```
let rec f g =
  let b1 = Get () in
  if g b1 then ()
  else
    let b2 = Get () in
    if b1 = b2 then f g
    else Raise ()
```

(a) The function `f` that uses the operations **Set**, **Get**, and **Raise** to invoke the effects.

```
(with {
  return x  ↦
    fun s -> (x,s),
  Set(x,k)  ↦
    fun s -> k () x,
  Get(_,k)  ↦
    fun s -> k s s,
  Raise(_,k) ↦
    fun s -> Raise ()
} handle (f v_g)) true
```

(b) Mutable state.

```
(with {
  return x  ↦
    fun s -> (x,s),
  Set(x,k)  ↦
    fun s -> k () s,
  Get(_,k)  ↦
    fun s -> k s s,
  Raise(_,k) ↦
    fun s -> Raise ()
} handle (f v_g)) true
```

(c) Immutable state.

Fig. 3. Examples where **Set** and **Get** are implemented by effect handlers.

Another way to define the semantics of algebraic effects is to use *effect handlers* [Plotkin and Pretnar 2009, 2013], which enable the user to alter the way effects are handled, thus being able to modify the meaning of algebraic effects during the program execution. For instance, consider the programs presented in Figure 3. Compared to Figure 1a, the function `f` in Figure 3a invokes the operations **Set**, **Get**, and **Raise** to represent the assignment `x := v`, dereference `!x`, and exception raising `raise ()`, respectively. An effect handler can define the meaning of these operations by being activated through a *handling construct*.

A handling construct `with` H `handle` e installs the effect handler H to interpret the effects invoked by the term e. If e calls an operation σ, its meaning is determined by the *operation clause* for σ given by H. An operation clause for σ takes the form $\sigma(\mathtt{x},\mathtt{k}) \mapsto e_\sigma$. Once e invokes σ, the handling construct is replaced by the clause's body e_σ where the variable `x` refers to the parameter of the operation call and the variable `k` refers to the *continuation* from the operation call up to the effect handler. For example, consider the program in Figure 3b, which implements mutable state in the state monad style; we refer to the effect handler there by H_{mut}. The function `f` first calls the operation **Get**. The functional representation of its continuation up to the effect handler is

```
fun b1 -> with H_mut handle
            if v_g b1 then ()
            else
              let b2 = Get () in
              if b1 = b2 then f v_g
              else Raise () .
```

Note that it corresponds to the continuation of the first call to **Get** in Figure 1b (except for the presence of the handling construct). Then, as the effect handler H_{mut} has the clause **Get**`(_,k)` $\mapsto$ `fun s -> k s s`, the handling construct is replaced by `fun s ->` v `s s` where v is the continuation described above. Since the handling construct is applied to the Boolean value `true`, the continuation v is called with `true` as the current (and initial) state of the reference. The clause for **Set** behaves similarly but it updates the state by a parameter `x`. The clause for **Raise** discards the continuation and forwards **Raise** to the outer context. For instance, if the function `f` is applied to the function `fun b -> Set (not b); false`, the program evaluation reaches **Raise** `()`. Because the effect handler forwards **Raise**, the evaluation stops at **Raise** `()`. Our tool successfully found this counterexample trace to the property that no exception should be raised, in 4 milliseconds on an ordinary laptop.

Once the term e in a handling construct `with` H `handle` e evaluates to a value v, the *return clause* of H is evaluated next. A return clause takes the form `return x ->` e_r, where the variable `x` refers to the value v of the term e. For instance, if the program in Figure 3b uses `fun b -> true` as v_g, it evaluates to `with` H_{mut} `handle ()`. Because the return clause of H_{mut} returns the pair of the evaluation result and the current state, the program evaluation results in the pair `((), true)`.

Although the effect handler H_{mut} implements mutable state, we can also implement *immutable* state using another effect handler as in Figure 3c. The effect handler there, to which we refer by H_{imm}, implements **Set** so that the new state is discarded and the old state is preserved (see the shaded part). Thus, it retains the initial state forever. Under this effect handler, the application `f (fun b -> Set (not b); false)` diverges without calling **Raise** because **Get** always return `true` and, hence, the test `b1 = b2` in the definition of `f` always succeeds. Furthermore, we can use different effect handlers at different places as `((with` H_{mut} `handle (f` v_g`)) true); ((with` H_{imm} `handle (f` v_g`)) true)` where the operator `;` is the sequential composition.

Dal Lago and Ghyselen [2024] give a λ-calculus extended with effect handlers, called HEPCF, and define its computation tree semantics, which is similar to that of EPCF except that a generated tree is composed only of *unhandled* algebraic operations.[3] For example, the computation tree of an HEPCF term with H_{mut} handle $f\ (\lambda b.\, \mathsf{let}\ b' = \mathsf{not}\ b\ \mathsf{in}\ \mathbf{Set}(b'; \_.\, \mathsf{false}))$ consists only of the root node **Raise**, and no node labeled **Set** nor **Get** appears because they are handled by H_{mut}. Note that **Raise** is also handled by H_{mut}, but it is forwarded. Thus, for the top-level context, it is regarded to be unhandled. From the viewpoint of the computation tree semantics, effect handlers can be considered as computation tree transformers [Pretnar 2015]. For example, recall that the term $f\ (\lambda b.\, \mathsf{let}\ b' = \mathsf{not}\ b\ \mathsf{in}\ \mathbf{Set}(b'; \_.\, \mathsf{false}))$ with the recursive function f defined in Figure 3a generates the tree in Figure 2c. The effect handler H_{mut} transforms it to the finite tree **Raise** according to its clauses, that is, by choosing the computation tree's paths that are consistent with the behavior of mutable state and eliminating **Set** and **Get** from the chosen paths (as they have been "handled").

Dal Lago and Ghyselen also showed the undecidability of HOMC for HEPCF terms by encoding natural numbers. Their idea of the encoding is to represent a natural number n as a term that calls some operation, say, **Succ**, exactly n times. Such a term generates the computation tree like

$$\mathbf{Succ} - \mathbf{Succ} - \cdots - \mathbf{Succ} - ()$$

with n nodes labeled **Succ** (the root node is the leftmost one). We refer to such a computation tree with n **Succ** by t_n. Then, we can provide an algebraic effect handler that transforms the computation tree t_{n+1} to t_n. By utilizing this idea, Dal Lago and Ghyselen succeeded in encoding the deconstructor of natural numbers. Because HEPCF allows an unbounded number of effect handlers to enclose a term at the same time, we can apply the encoded deconstructor an arbitrary number of times, which enables the full encoding of natural numbers.

2.3 Decidable Model Checking of Effect-Handling Programs by Answer-Type Modification

Because it turns out that HOMC for HEPCF is undecidable, a natural question that arises is what restriction on HEPCF terms makes the HOMC decidable while allowing a wide range of effects to be implemented by effect handlers. To answer this question, Dal Lago and Ghyselen provided GEPCF, a variant of HEPCF that only supports effect handlers in a restricted form, but it cannot

[3] As the target of the model checking is a generated computation tree, Dal Lago and Ghyselen's framework can only verify the properties of unhandled operations. To verify the properties of handled operations, one needs to make effect handlers forward the handled operations so that they appear in computation trees as unhandled ones or to use other verification methods [Gordon 2020; Kawamata et al. 2024; Sekiyama and Unno 2023; Song et al. 2022].

express some well-known effects such as mutable state, nondeterminism, and exception handling (see Section 4.4 for detail).

As a complementary approach, we restrict HEPCF terms through a type system that can bound the number of effect handlers made active at the same time. Namely, our type system only accepts terms such that, if they evaluate to a term

$$\cdots\ (\mathsf{with}\ H_1\ \mathsf{handle}\ \cdots\ (\mathsf{with}\ H_2\ \mathsf{handle}\ \cdots\ (\mathsf{with}\ H_n\ \mathsf{handle}\ e)\ \cdots)\ \cdots)\ \cdots$$

where only n effect handlers are active on the term e, the number n is bounded statically. This restriction leads to bounding a number of times an encoded natural number is deconstructed. Thus, terms can access only to a finite range of natural numbers, which results in restricting the available data domains to be *finite* and ensuring that the HOMC problem is decidable.

To achieve such a type system, we employ *answer-type modification* (ATM), which was introduced by Danvy and Filinski [1990] for the delimited control operator set shift/reset and adapted to effect handlers by Cong and Asai [2022] and Kawamata et al. [2024]; our type system is based on the one in the latter work. An *answer type* of a term is the type of the nearest delimiter—which corresponds to a handling construct in effect handlers—enclosing the term. For example, in the program

```
with {return x -> x, Ask(_, k) -> k true} handle let b = Ask () in not b,
```

the answer type of the operation call `Ask ()` is the Boolean type bool because the nearest handling construct returns a Boolean value.

One distinguished characteristic of ATM is the ability to allow answer types to be modified by the execution of effectful terms. This is a powerful, useful mechanism to track value flows [Kawamata et al. 2024] and traces [Sekiyama and Unno 2023].

However, another characteristic of ATM is crucial for our aim: it can bound the number of effect handlers made active at the same time. To highlight this point, in this section we adopt a simplified form of ATM, where answer types are not modified, contrary to the name; we will show a generic form to allow modification of answer types in Section 4. In the simplified ATM, we assign to a term a *computation type* of the form $\Sigma \triangleright T \,/\, A$, which is composed of three components: Σ is a signature of the operations performed by the term; T is the *value type* of the term, which specifies what value the term evaluates to (e.g., $T = \mathsf{bool}$ if the term evaluates to a Boolean); and A is the answer type of the term, which is either of a computation type or a value type. Thus, in general, a computation type can be described as $C \stackrel{\mathrm{def}}{=} \Sigma \triangleright T \,/\, (\Sigma_1 \triangleright T_1 \,/\, (\cdots (\Sigma_n \triangleright T_n \,/\, T_{n+1}) \cdots))$. For $i \in \{1, \cdots, n\}$, as $\Sigma_i \triangleright T_i \,/\, (\cdots (\Sigma_n \triangleright T_n \,/\, T_{n+1}) \cdots)$ is the i-th answer type, it is the type of the i-th nearest handling construct enclosing the effectful term of the type C. Thus, the term can be enclosed by n handling constructs and the enclosing term has the type $\Sigma_n \triangleright T_n \,/\, T_{n+1}$. Can we install more effect handlers on the enclosing term? No, it is invalid. If we were able to place the term under another handling construct, the handling construct would have the type T_{n+1}, but handling constructs are (possibly) effectful, so they cannot have value types like T_{n+1}. This indicates that, under ATM, only a statically bounded number of effect handlers can be active at the same time and that the counterexample to the decidability provided by Dal Lago and Ghyselen cannot be typechecked.

Our type system with ATM is expressive enough to accommodate effect handlers for mutable state, nondeterminism, and exception handling, as illustrated in Section 4.4. On the other hand, it does not accept all the well-typed GEPCF terms because GEPCF allows an unbounded number of effect handlers to be active at the same time, at the cost of the expressivity of effect handlers.

3 EPCF: PCF with Algebraic Effects

In this section, we introduce EPCF, a finitary variant of PCF with algebraic effects, and then formalize HOMC for EPCF as Dal Lago and Ghyselen [2024].

Variables x, y, z, f, h, k **Operations** σ

$$
\begin{array}{rrcl}
\textbf{Base constants} & c & ::= & \mathsf{true} \mid \mathsf{false} \mid () \mid \cdots \\
\textbf{Enum constants} & \varepsilon & ::= & \underline{1} \mid \underline{2} \mid \cdots \\
\textbf{Values} & v & ::= & x \mid c \mid \varepsilon \mid \lambda x.e \mid \mathsf{fix}\, x.v \\
\textbf{Terms} & e & ::= & \mathsf{return}\ v \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid v_1\ v_2 \mid \mathsf{case}(v; e_1, \cdots, e_n) \mid \sigma(v; x.\, e) \\
\textbf{Base types} & B & ::= & \mathsf{bool} \mid \mathsf{unit} \mid \cdots \\
\textbf{Enum types} & E & ::= & \mathsf{1} \mid \mathsf{2} \mid \cdots \\
\textbf{Types} & \tau & ::= & B \mid E \mid \tau_1 \to \tau_2 \\
\textbf{Operation signatures} & \Xi & ::= & \{\sigma_i : B_i \rightsquigarrow E_i\}^{1 \le i \le n} \\
\textbf{Typing contexts} & \Delta & ::= & \emptyset \mid \Delta, x : \tau
\end{array}
$$

Evaluation rules $\boxed{e_1 \longrightarrow e_2}$

$$
\begin{array}{rcl@{\qquad}rcl}
(\lambda x.e_1)\ v_2 & \longrightarrow & e_1[v_2/x] & (\mathsf{fix}\, x.v_1)\ v_2 & \longrightarrow & v_1[\mathsf{fix}\, x.v_1/x]\ v_2 \\
\mathsf{case}(\underline{i}; e_1, \cdots, e_n) & \longrightarrow & e_i \quad (\text{if } 0 < i \le n) & \mathsf{let}\ x = \mathsf{return}\ v_1\ \mathsf{in}\ e_2 & \longrightarrow & e_2[v_1/x] \\
\mathsf{let}\ x = \sigma(v_1; y.\, e_1)\ \mathsf{in}\ e_2 & \longrightarrow & \sigma(v_1; y.\, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \quad (\text{if } y \notin fv(e_2)) & & &
\end{array}
$$

$$
\frac{e_1 \longrightarrow e_1'}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \longrightarrow \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2}
$$

Typing rules $\boxed{\Xi \parallel \Delta \vdash v : \tau}$ $\boxed{\Xi \parallel \Delta \vdash e : \tau}$

$$
\frac{}{\Xi \parallel \Delta \vdash x : \Delta(x)} \qquad \frac{}{\Xi \parallel \Delta \vdash c : ty(c)} \qquad \frac{0 < i \le n}{\Xi \parallel \Delta \vdash \underline{i} : \mathsf{n}}
$$

$$
\frac{\Xi \parallel \Delta, x : \tau_1 \vdash e : \tau_2}{\Xi \parallel \Delta \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{\Xi \parallel \Delta, x : \tau_1 \to \tau_2 \vdash v : \tau_1 \to \tau_2}{\Xi \parallel \Delta \vdash \mathsf{fix}\, x.v : \tau_1 \to \tau_2}
$$

$$
\frac{\Xi \parallel \Delta \vdash v : \tau}{\Xi \parallel \Delta \vdash \mathsf{return}\ v : \tau} \qquad \frac{\Xi \parallel \Delta \vdash e_1 : \tau_1 \quad \Xi \parallel \Delta, x : \tau_1 \vdash e_2 : \tau_2}{\Xi \parallel \Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}
$$

$$
\frac{\Xi \parallel \Delta \vdash v_1 : \tau_1 \to \tau_2 \quad \Xi \parallel \Delta \vdash v_2 : \tau_1}{\Xi \parallel \Delta \vdash v_1\ v_2 : \tau_2} \qquad \frac{\Xi \parallel \Delta \vdash v : \mathsf{n} \quad \forall i \in [1, n].\ \Xi \parallel \Delta \vdash e_i : \tau}{\Xi \parallel \Delta \vdash \mathsf{case}(v; e_1, \cdots, e_n) : \tau}
$$

$$
\frac{\Xi \ni \sigma : B \rightsquigarrow E \quad \Xi \parallel \Delta \vdash v : B \quad \Xi \parallel \Delta, x : E \vdash e : \tau}{\Xi \parallel \Delta \vdash \sigma(v; x.\, e) : \tau}
$$

Fig. 4. The syntax, semantics, and type system of EPCF.

3.1 Definition and Basic Properties

The definition of EPCF is shown in Figure 4.

The program syntax of EPCF consists of two syntactic categories: *values* and *terms*. Values, ranged over by v, syntactically represent a canonical form not to be evaluated further, composed of variables, base constants, enum constants, functions $\lambda x.e$ (x is bound in e), and the fixpoint operator $\mathsf{fix}\, x.v$ (x is bound in v) on functions. Base constants, ranged over by c, are primitive values, and enum constants, ranged over by ε, allow the user to make case analysis on them. Terms, ranged over by e, may involve possibly effectful computation. Value-return constructs $\mathsf{return}\ v$, let expressions $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ (x is bound in e_2), function applications $v_1\ v_2$ are standard. A case expression $\mathsf{case}(v; e_1, \cdots, e_n)$ performs case analysis on enum constants: if the value v is an enum

constant $\underline{i}$ for some i such that $0 < i \leq n$, the case expression reduces to e_i. It can be easily seen that some constructs used in Section 2.1, such as if–then constructs and absurd, can be expressed using case expressions. An algebraic operation call (operation call for short) $\sigma(v; x.\, e)$ involves a parameter value v and a continuation of the form $x.\, e$ (x is bound in e), which represents the remaining computation after σ determines the return value denoted by x.

The set of free variables in a term e, denoted by $fv(e)$, is defined in a standard manner. We also write $e[v/x]$ and $v'[v/x]$ for the term and value obtained by substituting the value v for x in the term e and value v', respectively, in a capture-avoiding manner. We use the similar notation for $\mathsf{HEPCF}^{\mathsf{ATM}}$ defined in Section 4.

The operational semantics of EPCF is defined by the evaluation relation $\longrightarrow$, which is the smallest binary relation over terms that satisfies the rules presented in Figure 4. Most of the rules are standard or formalize the behavior of terms as explained. It should be noted that, if an algebraic operation call is followed by another term as $\mathsf{let}\, x = \sigma(v; y.\, e_1)\, \mathsf{in}\, e_2$, the term e_2 is incorporated into the continuation $y.\, e_1$ attached to the operation call because e_2 is the remaining computation after the continuation e_1. As a result, the let expression reduces to $\sigma(v; y.\, \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2)$.

Types, ranged over by τ, are base types B for base constants, enum types E for enum constants, or function types. We assume that every base type only has a finite number of inhabitants.[4] An enum type takes the form n for some natural number n, which has n inhabitants $\underline{1}, \cdots, \underline{n}$. The signature of operations is determined by an operation signature Ξ, which maps operations to their types of the form $B \rightsquigarrow E$. When $B \rightsquigarrow E$ is assigned to σ, the term $\sigma(v; x.\, e)$ is typechecked if the parameter v is of the type B—thus, B is called the *parameter type* of σ—and e is typed under the assumption that x has the type E. If $E = \mathsf{n}$ for some n, the continuation $x.\, e$ is identified with n terms $e[\underline{1}/x], \cdots, e[\underline{n}/x]$. Thus, we can deem the arity of σ to be n, so E is called the *arity type* of σ. Restricting arity types to be enum types enables us to interpret an EPCF term to be a possibly infinite tree with finite branches, as the ones generated by HORS, and makes the HOMC problem for it decidable. Typing contexts, ranged over by Δ, are finite sequences of bindings of variables coupled with types. We assume that the types assigned to the same variable by Δ can be uniquely determined (namely, for any x, τ, and τ', $x : \tau \in \Delta$ and $x : \tau' \in \Delta$ imply $\tau = \tau'$). We write $\Delta(x)$ for the type assigned to x by Δ; it is undefined if Δ includes no binding for x.

The type system gives inference rules to derive typing judgments of the form $\Xi \parallel \Delta \vdash v : \tau$ for values and $\Xi \parallel \Delta \vdash e : \tau$ for terms. It is standard and should be self-explanatory. We assume a function ty that assigns a base type to every base constant.

Finally, we show the basic properties of EPCF: progress, subject reduction, and determinacy.

Lemma 1 (Progress). *If $\Xi \parallel \emptyset \vdash e : \tau$, then one of the following holds: $e = \mathsf{return}\, v$ for some v; $e = \sigma(v; x.\, e')$ for some σ, v, x, and e'; or $e \longrightarrow e'$ for some e'.*

Lemma 2 (Subject Reduction). *If $\Xi \parallel \Delta \vdash e : \tau$ and $e \longrightarrow e'$, then $\Xi \parallel \Delta \vdash e' : \tau$.*

Lemma 3 (Determinacy). *If $e \longrightarrow e_1$ and $e \longrightarrow e_2$, then $e_1 = e_2$.*

3.2 Model Checking

This section recaps HOMC for EPCF, illustrating some examples of the HOMC after formalizing it.

3.2.1 Definition and Properties. The HOMC problem for EPCF is defined using *effect trees*, which are structures to be model checked, and *alternating parity tree automata (APTs)*, which specify the behavior of effects and properties to be verified.

[4] Under this assumption, enum constants include base constants, but Dal Lago and Ghyselen [2024] distinguish between them to identify where the case split, which is allowed only for enum constants, plays a critical role [Dal Lago 2024].

Effect trees are defined for well-typed EPCF terms. They are similar to computation trees presented in Section 2.1 but may involve a new tree constructor $\bot$ to represent the divergence. Hereinafter, we use the following notation:

$$\begin{aligned} e \longrightarrow^n e' &\stackrel{\text{def}}{=} \exists e_0, \cdots, e_n.\, e = e_0 \wedge (\forall i < n.\, e_i \longrightarrow e_{i+1}) \wedge e_n = e', \\ e \longrightarrow^* e' &\stackrel{\text{def}}{=} \exists n.\, e \longrightarrow^n e', \text{ and} \\ e \longrightarrow^\omega &\stackrel{\text{def}}{=} \forall n.\, \exists e'.\, e \longrightarrow^n e'. \end{aligned}$$

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 $ar_S(s)$ 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):*

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

Definition 3 (Effect Trees for EPCF Terms). *Given an operation signature Ξ and a type τ, the tree constructor signature S^Ξ_τ is defined as follows:*

$$S^\Xi_\tau \stackrel{\text{def}}{=} \{\sigma : n+1 \mid \sigma : B \rightsquigarrow \mathsf{n} \in \Xi\} \cup \{\mathsf{return}\ v : 0 \mid \Xi \parallel \emptyset \vdash v : \tau\} \cup \bigcup_c \{c : 0\}$$

where, for a tree constructor s (that is an operation σ, return construct return *v, or base constant c), $s : n$ denotes the pair (s, n), meaning that the arity of s is n. Given a term e such that $\Xi \parallel \emptyset \vdash e : \tau$, the effect tree of e, denoted by $\mathbf{ET}(e)$, is a tree in $\mathbf{Tree}_{S^\Xi_\tau}$ defined by the following (possibly infinite) process:*

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

Any well-typed EPCF term has a uniquely determined effect tree.

Lemma 4 (Well-Definedness of EPCF Effect Trees). *If $\Xi \parallel \emptyset \vdash e : \tau$, then $\mathbf{ET}(e)$ is well defined and uniquely determined, and it is in $\mathbf{Tree}_{S^\Xi_\tau}$.*

Next, we introduce APTs along with auxiliary notions.

Definition 4 (Positive Boolean Formulas). *The set $\mathbf{B}^+(X)$ of* positive Boolean formulas *over a finite set X is defined as follows:*

$$\mathbf{B}^+(X) \ni \theta ::= \mathsf{tt} \mid \mathsf{ff} \mid \mathbf{x} \mid \theta_1 \vee \theta_2 \mid \theta_1 \wedge \theta 2$$

where $\mathbf{x} \in X$. A subset Y of X satisfies $\theta \in \mathbf{B}^+(X)$ if θ holds under the interpretation that assigns true to the elements in Y and false to the elements in $Y \setminus X$.

Definition 5 (Positions of Trees). *The set $dom(t)$ of the* positions *of a tree t generated by a tree constructor signature S with maximal arity n is a set of finite sequences over alphabet $\{1, \cdots, n\}$ defined as $dom(\bot) = \{\epsilon\}$ and $dom(s(t_1, \cdots, t_{ar_S(s)})) = \{\epsilon\} \cup \bigcup_{i \in \{1, \cdots, ar_S(s)\}} \{i \cdot p \mid p \in dom(t_i)\}$, where ϵ is the empty sequence and $\cdot$ is the concatenation of finite sequences.*

The node $t(p)$ of a tree t at a position $p \in dom(t)$ is defined by $\bot(\epsilon) = \bot$, $s(t_1, \cdots, t_{ar_S(s)})(\epsilon) = s$, and $s(t_1, \cdots, t_{ar_S(s)})(i \cdot p) = t_i(p)$.

Definition 6 (Alternating Parity Tree Automata). *An* alternating parity tree automaton *(APT) over a tree constructor signature S is a tuple $\mathcal{A} = (S, Q, \delta, q_I, \Omega)$ satisfying the following:*

- Q *is a finite set of states with* $q_I \in Q$ *as the initial state.*
- δ *is a transition function, mapping* $(q, s) \in Q \times dom(S)$ *to a formula in* $\mathbf{B}^+(\{1, \cdots, ar_S(s)\} \times Q)$.
- Ω *is a priority function, mapping states in* Q *to natural numbers.*

A run-tree *of an APT* $\mathcal{A} = (S, Q, \delta, q_I, \Omega)$ *over a tree* $t \in \mathbf{Tree}_S$ *is a tree satisfying the following:*

- *Every node is labeled with some* $(p, q) \in dom(t) \times Q$.
- *The root node is* (ϵ, q_I).
- *For each node* (p, q)*, there is a set* $X \subseteq \{1, \cdots, ar_S(t(p))\} \times Q$ *satisfying the positive Boolean formula* $\delta(q, t(p))$ *and, for each* $(i, q') \in X$*, the node* $(p \cdot i, q')$ *is a child of the node* (p, q).

A tree $t \in \mathbf{Tree}_S$ *is* accepted *by an APT* $\mathcal{A}$ *if there exists a run-tree of* $\mathcal{A}$ *over* t *such that every infinite path* $(\epsilon, q_I), (p_1, q_1), (p_2, q_2) \cdots$ *of the run-tree meets the parity condition, that is, the largest priority infinitely occurring in* $\Omega(q_I), \Omega(q_1), \Omega(q_2), \cdots$ *is even.*

Finally, we define the HOMC problem for EPCF. A type τ is *ground* if it is a base or enum type.

Definition 7 (Higher-Order Model Checking Problem for EPCF). *Given an APT and an* EPCF *term* e *such that* $\Xi \parallel \emptyset \vdash e : \tau$ *for some* Ξ *and ground type* τ*, is* $\mathbf{ET}(e)$ *accepted by the APT?*

Theorem 1 (Decidability of Model Checking for EPCF [Dal Lago and Ghyselen 2024]). *The higher-order model checking problem for* EPCF *is decidable.*

3.2.2 Examples. We present two examples of HOMC, where APTs describe specifications to be verified and specify the behavior intended on algebraic effects.

Example 3.1. Let Ξ be an operation signature for **Set**, **Get**, and **Raise**, that is,

$$\Xi \stackrel{\text{def}}{=} \{\mathbf{Set} : \mathsf{bool} \rightsquigarrow 1, \mathbf{Get} : \mathsf{unit} \rightsquigarrow 2, \mathbf{Raise} : \mathsf{unit} \rightsquigarrow 0\}\ .$$

Let $\mathcal{A}_{\mathbf{SGR}} \stackrel{\text{def}}{=} (S^{\Xi}_{\mathsf{unit}}, Q, \delta, q_1, \{q \mapsto 0 \mid q \in Q\})$ where $Q = \{q_1, q_2, q_{\mathsf{true}}, q_{\mathsf{false}}\}$ and δ is defined as

- $\delta(q_i, \mathbf{Set}) = ((1, q_{\mathsf{true}}) \wedge (2, q_1)) \vee ((1, q_{\mathsf{false}}) \wedge (2, q_2))$,
- $\delta(q_i, \mathbf{Get}) = (i + 1, q_i)$,
- $\delta(q_i, \mathbf{Raise}) = \mathsf{ff}$,
- $\delta(q_i, \mathsf{return}\ v) = \delta(q_b, b) = \mathsf{tt}$ and $\delta(q_b, b') = \mathsf{ff}$

for each $i \in \{1, 2\}$, $b \in \{\mathsf{true}, \mathsf{false}\}$, and $b' \in \{\mathsf{true}, \mathsf{false}\} \setminus \{b\}$ (in the other cases, δ returns ff).

The states q_1 and q_2 express the program states where the global reference manipulated by **Set** and **Get** refers to true and false, respectively. Based on this idea, the transition rules for **Set** and **Get** encode the semantics of mutable state in the APT: if **Set** is called with the parameter true (resp. false), the continuation is executed under the state q_1 (resp. q_2); if **Get** is called in the state q_1 (resp. q_2), the continuation supposing the return value of **Get** to be $\underline{1}$ (resp. $\underline{2}$) is chosen. The conjunct $(1, q_{\mathsf{true}})$ (resp. $(1, q_{\mathsf{false}})$) in the transition of **Set** requires that the parameter of **Set** be true (resp. false) to set the state of the continuation to q_1 (resp. q_2).

The transition rule for **Raise** expresses the specification that **Raise** must not be called. This is indicated by the fact that we cannot make a run-tree of $\mathcal{A}_{\mathbf{SGR}}$ over a tree t that involves a path where a **Raise** node is reachable and **Set** and **Get** interact in accordance with the semantics of mutable state (for instance, Figure 2c shows an example of such a tree). If there exists such a run-tree, it would contain a node (p, q_i) for some $p \in dom(t)$ and $i \in \{1, 2\}$ such that $t(p) = \mathbf{Raise}$. By the definition of run-trees, there should be some X that satisfies the positive Boolean formula $\delta(q_i, \mathbf{Raise})$, but there is no such X because $\delta(q_i, \mathbf{Raise}) = \mathsf{ff}$. Thus, there is no run-tree of $\mathcal{A}_{\mathbf{SGR}}$ over the tree t, which means that the tree t is not accepted by $\mathcal{A}_{\mathbf{SGR}}$.

For instance, the effect trees in Figures 2a and 2b are accepted by $\mathcal{A}_{\mathbf{SGR}}$, but the one in Figure 2c is not. Note that the tree in Figure 2b involves paths to reach **Raise**, but there **Get** does not conform to the semantics of mutable state, as consecutive calls to **Get** return different Boolean values.

Example 3.2. Consider verifying the use of global file manipulation operations **Open** that opens the file, **Read** that reads the contents of the opened file, and **Close** that closes the opened file. The use of these operations is valid if their call sequences conform to the regular expression $(\mathbf{Open}\,\mathbf{Read}^*\,\mathbf{Close})^*$. Let Ξ be an operation signature for the file operations defined as:

$$\Xi \stackrel{\text{def}}{=} \{\mathbf{Open} : \mathsf{unit} \rightsquigarrow 1, \mathbf{Read} : \mathsf{unit} \rightsquigarrow 2, \mathbf{Close} : \mathsf{unit} \rightsquigarrow 1\}\ .$$

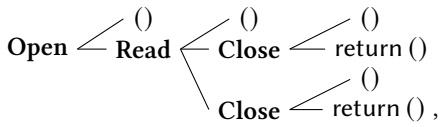
Here, we assume simpler file operations than the operations in practice (as in POSIX): **Open** does not take a file path nor return a file descriptor to identify the opened file object (thus, the file to be manipulated is predetermined) and **Read** only returns enum constant $\underline{1}$ or $\underline{2}$. Nevertheless, it is still nontrivial to verify the valid use of the file operations even for this simplified version. We can treat more practical file operations, such as ones that can manipulate multiple files, by adapting the techniques in the previous work [Kobayashi 2013].

An APT $\mathcal{A}_{\mathbf{File}}$ that only accepts effect trees where file operations are used in a valid manner is given by $(S^{\Xi}_{\tau}, \{q_1, q_2\}, \delta, q_1, \{q_1 \mapsto 2, q_2 \mapsto 1\})$ where the transition function δ is defined by

- $\delta(q_1, \mathbf{Open}) = (2, q_2)$ and $\delta(q_2, \mathbf{Open}) = \mathsf{ff}$,
- $\delta(q_2, \mathbf{Read}) = (2, q_2) \wedge (3, q_2)$ and $\delta(q_1, \mathbf{Read}) = \mathsf{ff}$,
- $\delta(q_2, \mathbf{Close}) = (2, q_1)$ and $\delta(q_1, \mathbf{Close}) = \mathsf{ff}$,
- $\delta(q_1, \mathsf{return}\ v) = \mathsf{tt}$ and $\delta(q_2, \mathsf{return}\ v) = \mathsf{ff}$, and
- $\delta(q_1, c) = \delta(q_2, c) = \mathsf{tt}$

(the type τ is of EPCF terms to be verified). The state q_1 and q_2 represent that the file is closed and opened, respectively. Thus, a call to **Open** in the state q_1 and a call to **Read** or **Close** in the state q_2 are valid, whereas a call to **Open** in q_2 and a call to **Read** or **Close** in q_1 are invalid. Furthermore, the valid call to **Open** or **Close** flips the state. The transition of **Read** at the state q_2 expresses that **Read** returns $\underline{1}$ or $\underline{2}$ nondeterministically. The transition of return v at q_2 means that a term to be verified must not terminate in the state that the file are left open.

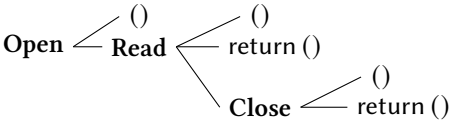
For example, a term $\mathbf{Open}((); x.\, \mathbf{Read}((); y.\, \mathbf{Close}((); z.\, \mathsf{return}\ ())))$ generates the effect tree t as



and a run-tree over t can be given as

$$\begin{array}{l}(\epsilon, q_1) \text{ — } (2, q_2) \text{ — } (2 \cdot 2, q_2) \text{ — } (2 \cdot 2 \cdot 2, q_1) \\ \qquad\qquad\quad\ \ \text{ — } (2 \cdot 3, q_2) \text{ — } (2 \cdot 3 \cdot 2, q_1)\ .\end{array}$$

For a term $\mathbf{Open}((); x.\, \mathbf{Read}((); y.\, \mathsf{case}(y; \mathsf{return}\ (), \mathbf{Close}((); z.\, \mathsf{return}\ ()))))$, the effect tree t' like



is generated, but we cannot make a run-tree over t'; the process to make a run-tree will stop at

$$\begin{array}{l}(\epsilon, q_1) \text{ — } (2, q_2) \\ \qquad\qquad\quad\ \ \text{ — } (2 \cdot 3, q_2) \text{ — } (2 \cdot 3 \cdot 2, q_1)\ ,\end{array}$$

but there is no finite set X satisfying $\delta(q_2, t'(2 \cdot 2)) = \delta(q_2, \mathsf{return}\ ()) = \mathsf{ff}$.

The priority function ensures that the opened file will be closed eventually for divergent programs. To see it, consider the effect tree t generated by a term $\mathbf{Open}((); x.\ (\mathsf{fix}\, f.\lambda y.\mathbf{Read}((); z.\ f\ y))\ ())$, which infinitely calls **Read** after performing **Open**. Given a run-tree over t, it should include an infinite path where q_2 occurs infinitely and q_1 occurs only at the root node. The largest priority infinitely occurring in such a path is 1, which results in breaking the parity condition. Thus, the effect tree t is not accepted by $\mathcal{A}_{\mathbf{File}}$. Note that, if the opened file is eventually closed as in a term $(\mathsf{fix}\, f.\lambda x'.\mathbf{Open}((); x.\ \mathbf{Read}((); y.\ \mathbf{Close}((); z.\ f\ x'))))\ ()$, we can make a run-tree where the state q_1 appears infinitely many times in every infinite path, so the largest priority infinitely occurring in an infinite path is even and the parity condition holds.

Remark. To verify a term e that may diverge without performing effect operations (e.g., it may only call recursive functions infinitely), we would need to insert a "dummy effect" that happens every time a recursive function is called. If there is no such a dummy effect, the effect tree t generated by the term e may involve the leaf $\bot$. However, we cannot make a run-tree with a node (q, p) such that $t(p) = \bot$ because then $ar_S(t(p))$ is undefined although it is required to be defined to make such a run-tree (or, more intuitively, because an APT is not aware of $\bot$). This limitation in run-trees would hinder the verification of the term e. The insertion of the dummy effect enables eliminating $\bot$ from effect trees.

4 HEPCF$^{\mathsf{ATM}}$: PCF with Answer-Type Modification for Algebraic Effects and Handlers

This section introduces HEPCF$^{\mathsf{ATM}}$, a λ-calculus with algebraic effects and handlers. The program syntax and semantics of HEPCF$^{\mathsf{ATM}}$ is the same as those of HEPCF given by Dal Lago and Ghyselen [2024]. Its type system allows ATM as that of Kawamata et al. [2024], but they differ in the type assignment to pure computations, which influences whether parametric polymorphism is required to define a type-preserving CPS transformation. Because the support for parametric polymorphism makes the HOMC problem undecidable [Tsukada and Kobayashi 2010], we adapt Kawamata et al.'s type system so that a CPS transformation for HEPCF$^{\mathsf{ATM}}$ can be defined without relying on parametric polymorphism. We will come back to this point in Section 5.3.

4.1 Program Syntax

The program syntax of HEPCF$^{\mathsf{ATM}}$, which is presented in Figure 5, is similar to that of EPCF; we use the metavariables M and V to range over terms and values, respectively, in HEPCF$^{\mathsf{ATM}}$. The only syntactic difference from EPCF is the support for *handling constructs* of the form $\mathsf{with}\ H\ \mathsf{handle}\ M$, which behaves as explained in Section 2.2; we call the term M a *handled term*. We assume that any effect handler, denoted by H, has at most one operation clause for each algebraic operation. As in EPCF, we denote the set of free variables in a term M by $fv(M)$ and value substitution for variables in terms and values by $M[V/x]$ and $V'[V/x]$, respectively.

4.2 Semantics

The semantics of HEPCF$^{\mathsf{ATM}}$ is defined by the evaluation relation $\longrightarrow$, which is the smallest binary relation over terms that satisfies the rules in Figure 5. Again, most of the rules are similar to those of EPCF, and the exception is only the rules for handling constructs. Once an algebraic operation is called, it bubbles up to the nearest handling construct enclosing the call step by step. When it reaches the handling construct with an effect handler H, the corresponding operation clause in H is executed with the parameter and the continuation of the call. Note that the continuation passed to the operation clause is a function of the form $\lambda y.\mathsf{with}\ H\ \mathsf{handle}\ M$, which encloses the body M of the continuation attached to the operation call by the effect handler H. Effect handlers behaving in this way are called *deep*, and effect handlers that do not enclose the body of the continuation

$$
\begin{array}{rrcl}
\textbf{Values} & V & ::= & x \mid c \mid \varepsilon \mid \lambda x.M \mid \mathsf{fix}\, x.V \\
\textbf{Terms} & M & ::= & \mathsf{return}\, V \mid \mathsf{let}\, x = M_1\, \mathsf{in}\, M_2 \mid V_1\, V_2 \mid \mathsf{case}(V; M_1, \cdots, M_n) \mid \\
 & & & \sigma(V; x.\, M) \mid \mathsf{with}\, H\, \mathsf{handle}\, M \\
\textbf{Handlers} & H & ::= & \{\mathsf{return}\, x \mapsto M\} \uplus \{\sigma_i(x_i; k_i) \mapsto M_i\}^{1 \le i \le n}
\end{array}
$$

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

$$
\begin{array}{rcll}
(\lambda x.M_1)\, V_2 & \longrightarrow & M_1[V_2/x] & \\
(\mathsf{fix}\, x.V_1)\, V_2 & \longrightarrow & V_1[\mathsf{fix}\, x.V_1/x]\, V_2 & \\
\mathsf{case}(\underline{i}; M_1, \cdots, M_n) & \longrightarrow & M_i & (\text{if } 0 < i \le n) \\
\mathsf{let}\, x = \mathsf{return}\, V_1\, \mathsf{in}\, M_2 & \longrightarrow & M_2[V_1/x] & \\
\mathsf{let}\, x = \sigma(V_1; y.\, M_1)\, \mathsf{in}\, M_2 & \longrightarrow & \sigma(V_1; y.\, \mathsf{let}\, x = M_1\, \mathsf{in}\, M_2) & (\text{if } y \notin fv(M_2)) \\
\mathsf{with}\, H\, \mathsf{handle}\, \mathsf{return}\, V & \longrightarrow & M[V/x] & (\text{if } \mathsf{return}\, x \mapsto M \in H) \\
\mathsf{with}\, H\, \mathsf{handle}\, \sigma(V; y.\, M) & \longrightarrow & M'[V/x][\lambda y.\mathsf{with}\, H\, \mathsf{handle}\, M/k] & (\text{if } \sigma(x; k) \mapsto M' \in H)
\end{array}
$$

$$
\frac{M_1 \longrightarrow M_1'}{\mathsf{let}\, x = M_1\, \mathsf{in}\, M_2 \longrightarrow \mathsf{let}\, x = M_1'\, \mathsf{in}\, M_2}
\qquad
\frac{M \longrightarrow M'}{\mathsf{with}\, H\, \mathsf{handle}\, M \longrightarrow \mathsf{with}\, H\, \mathsf{handle}\, M'}
$$

Fig. 5. The program syntax and semantics of HEPCF$^{\mathsf{ATM}}$.

passed to an operation clause are called *shallow* [Kammar et al. 2013]. The difference between deep and shallow effect handlers influences the design of type systems with ATM, as discussed by Kawamata et al. [2024]. If the handled term of a handling construct returns a value, the return clause given by the effect handler is executed.

4.3 Type System

We show the type language of HEPCF$^{\mathsf{ATM}}$ at the top of Figure 6. Value types, ranged over by T, are types assigned to values, being base types, enum types, or function types.

Computation types, ranged over by C, are types assigned to terms, taking the form $\Sigma \triangleright T \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}}$. As explained in Section 2.3, Σ is an operation signature that determines the type interface of algebraic operations that the terms may call, and T is a value type that specifies values that the terms may return. The last two components A^{ini} and A^{fin} are *answer types*, which are either of value or computation types. Especially, A^{ini} is called an *initial* answer type and A^{fin} is called a *final* one. As seen in Section 2.3, answer types are the types of the nearest handling constructs enclosing the terms. A key difference from the formalism given in Section 2.3 is that answer types in HEPCF$^{\mathsf{ATM}}$ can be *modified* from the initial answer type A^{ini} to the final one A^{fin}; in fact, a computation type $\Sigma \triangleright T \,/\, A$ presented in Section 2.3 is an abbreviation of $\Sigma \triangleright T \,/\, A \Rightarrow A$. To see the expressivity of the modification in detail, consider a handling construct $\mathsf{with}\, H\, \mathsf{handle}\, \sigma(\underline{1}; y.\, \mathsf{return}\, y)$ with an effect handler $H \stackrel{\mathrm{def}}{=} \{\mathsf{return}\, x \mapsto \mathsf{return}\, x\} \uplus \{\sigma(x; k) \mapsto \mathsf{return}\, k\}$. For the operation call $\sigma(\underline{1}; y.\, \mathsf{return}\, y)$, the initial answer type represents what computation the continuation of the operation call up to the nearest enclosing handling construct performs. The continuation is $\mathsf{with}\, H\, \mathsf{handle}\, \mathsf{return}\, [\,]$ (where $[\,]$ represents the "hole" to be filled when the continuation is invoked) and it returns the value passed to fill the hole. Thus, if the hole is supposed to be filled with, say, enum constants of some type E, the initial answer type of the operation call is a computation type C^{ini} whose value type is E. When we take a look at the state after the operation call, the handling construct evaluates to the body of σ's clause in H, that is, to return v where v is $\lambda y.\mathsf{with}\, H\, \mathsf{handle}\, \mathsf{return}\, y$, the functional representation of the continuation. The final answer type A^{fin} of the operation call represents this result. More generally, the final answer type of a term M specifies terms to which

$$
\begin{array}{rrcl}
\textbf{Value types} & T & ::= & B \mid E \mid T \to C \\
\textbf{Computation types} & C & ::= & \Sigma \triangleright T \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}} \\
\textbf{Operation signatures} & \Sigma & ::= & \{\sigma_i : T_i^{\mathrm{par}} \rightsquigarrow T_i^{\mathrm{ari}} \,/\, A_i^{\mathrm{ini}} \Rightarrow A_i^{\mathrm{fin}}\}^{1 \le i \le n} \\
\textbf{Answer types} & A & ::= & T \mid C \\
\textbf{Typing contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x : T
\end{array}
$$

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

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

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

$$\frac{\Gamma \vdash V : T}{\Gamma \vdash \mathsf{return}\, V : \Sigma \triangleright T \,/\, A \Rightarrow A}\ \mathrm{HT\_Return}$$

$$\frac{\Gamma \vdash M_1 : \Sigma \triangleright T_1 \,/\, A \Rightarrow A_1 \quad \Gamma, x : T_1 \vdash M_2 : \Sigma \triangleright T_2 \,/\, A_2 \Rightarrow A}{\Gamma \vdash \mathsf{let}\, x = M_1 \,\mathsf{in}\, M_2 : \Sigma \triangleright T_2 \,/\, A_2 \Rightarrow A_1}\ \mathrm{HT\_Let}$$

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

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

$$\frac{\begin{array}{c} H = \{\mathsf{return}\, x \mapsto M_0\} \uplus \{\sigma_i(x_i; k_i) \mapsto M_i\}^{1 \le i \le n} \\ \Sigma = \{\sigma_i : T_i^{\mathrm{par}} \rightsquigarrow T_i^{\mathrm{ari}} \,/\, C_i^{\mathrm{ini}} \Rightarrow C_i^{\mathrm{fin}}\}^{1 \le i \le n} \qquad \Gamma \vdash M : \Sigma \triangleright T \,/\, C^{\mathrm{ini}} \Rightarrow C^{\mathrm{fin}} \\ \Gamma, x : T \vdash M_0 : C^{\mathrm{ini}} \quad \forall i \in [1, n].\, \Gamma, x_i : T_i^{\mathrm{par}}, k_i : T_i^{\mathrm{ari}} \to C_i^{\mathrm{ini}} \vdash M_i : C_i^{\mathrm{fin}} \end{array}}{\Gamma \vdash \mathsf{with}\, H \,\mathsf{handle}\, M : C^{\mathrm{fin}}}\ \mathrm{HT\_Handle}$$

Fig. 6. The type language and type system of HEPCF$^{\mathsf{ATM}}$.

the nearest handling construct enclosing the term M finally evaluates. Thus, A^{fin} is a computation type C^{fin} whose value type is $E \to C^{\mathrm{ini}}$. In summary, if the arity type of σ is E, the computation type of the operation call $\sigma(\underline{1}; y.\, \mathsf{return}\, y)$ can be $\Sigma \triangleright E \,/\, C^{\mathrm{ini}} \Rightarrow C^{\mathrm{fin}}$ for some Σ, C^{ini}, and C^{fin} such that the value types of C^{ini} and C^{fin} are E and $E \to C^{\mathrm{ini}}$, respectively.

A change from the type language of Kawamata et al. [2024] is that we disallow a computation type of the form $\Sigma \triangleright T \,/\, \Box$, where $\Box$, called the *pure control effect*, means that a term of this type calls no algebraic operation. We decided to get rid of the pure control effect because it can make an unbounded number of effect handlers active at the same time. In Kawamata et al.'s work, a computation type $\Sigma \triangleright T \,/\, \Box$ can be coerced into $\Sigma \triangleright T \,/\, (\Sigma_1 \triangleright T_1 \,/\, \Box) \Rightarrow (\Sigma_1 \triangleright T_1 \,/\, \Box)$ for some operation signature Σ_1 and value type T_1. Furthermore, the coerced type can be coerced into $\Sigma \triangleright T \,/\, (\Sigma_1 \triangleright T_1 \,/\, \Box) \Rightarrow (\Sigma_1 \triangleright T_1 \,/\, (\Sigma_2 \triangleright T_2 \,/\, \Box \Rightarrow \Sigma_2 \triangleright T_2 \,/\, \Box))$. We can repeat this process an unbounded number of times *where necessary*, and therefore any number of effect handlers can be installed on a term of $\Sigma \triangleright T \,/\, \Box$ at the same time. The support for the pure control effect influences the definition of CPS transformation. The CPS transformation given by Kawamata et al. relies on parametric polymorphism to address the pure control effect. This is inconvenient for our aim because parametric polymorphism invalidates the decidability of HOMC [Tsukada and Kobayashi

2010]. Excluding the pure control effect enables us to define a CPS transformation that does not rest on parametric polymorphism.[5] Instead, in our type language, answer types include value types to describe computation types in a finite, inductive manner. The type system of HEPCF$^{\mathsf{ATM}}$ allows only a term of a computation type of the form $\Sigma \triangleright T \,/\, C^{\mathrm{ini}} \Rightarrow C^{\mathrm{fin}}$ (that is, a computation type whose answer types are *not* value types) to be handled. This restriction makes it possible to uniformly require the return clause of an effect handler to be a computation (of the type C^{ini}) and ensure that a handling construct is also a computation (of the type C^{fin}). Any operation called by a term of a computation type, say, $\Sigma \triangleright T \,/\, T^{\mathrm{ini}} \Rightarrow T^{\mathrm{fin}}$ is never handled and, thus, it is regarded as an algebraic operation of a primitive effect.

Operation signatures, ranged over by Σ, determine the type interface of algebraic operations as those in EPCF, but there are two key differences between them. First, the parameter and arity types of an operation can be arbitrary in HEPCF$^{\mathsf{ATM}}$, whereas they are restricted to base and enum types, respectively, in EPCF. This is because all the operation calls in EPCF are unhandled and appears in computation trees. We allow the parameter and arity types of *handled* operations to be arbitrary and restrict those of *unhandled* operations in defining the effect trees of HEPCF$^{\mathsf{ATM}}$ terms. The second difference is that a type interface for an algebraic operation in HEPCF$^{\mathsf{ATM}}$ involves initial and final answer types assigned to calls to the operation.

Typing contexts, ranged over by Γ, are finite sequences of bindings of variables associated with value types. Note that only values are substituted for variables in HEPCF$^{\mathsf{ATM}}$.

The type system of HEPCF$^{\mathsf{ATM}}$ is shown at the bottom of Figure 6, equipped with inference rules to derive judgments $\Gamma \vdash V : T$ for values and $\Gamma \vdash M : C$ for terms. The rules for value typing judgments are similar to those in EPCF.

The typing rules for terms are an adaption of the rules given by Kawamata et al. [2024] to our setting. It is noteworthy that the same initial and finial answer types are assigned to a value-return construct. This reflects that Kawamata et al.'s type system assigns the pure control effect $\Box$ to a value-return construct and $\Box$ can be coerced into $C \Rightarrow C$ for any type C. For a let expression $\mathsf{let}\, x = M_1 \,\mathsf{in}\, M_2$, its initial and finial answer types are determined by the terms M_2 and M_1, respectively. This can be understood as follows. First, how the nearest handling construct is transformed to another term is determined by the term M_1 because the transformation depends on which operation M_1 calls. Thus, the final answer type is determined by M_1 (if M_1 calls no operation, then $A_1 = A$ and thus the final answer type A_1 is determined by M_2). Second, the continuation involving the nearest handling construct is finally captured by the term M_2. Thus, the initial answer type is determined by M_2. Note that, for a handling construct with H handle $\mathsf{let}\, x = M_1 \,\mathsf{in}\, M_2$, if M_1 is an algebraic operation call, it captures the continuation like with H handle $\mathsf{let}\, x = \mathsf{return}\ [\,]\ \mathsf{in}\, M_2$, where the nearest handling construct remains, but it may be transformed by the term M_2 when invoked. Thus, what can be ensured is only that M_2 may capture the continuation involving the nearest handling construct. The rule (HT_Op) for operation calls might look tricky, but it would be easily seen by considering $\sigma(V; x.\, M)$ as $\mathsf{let}\, x = \sigma(V; y.\, \mathsf{return}\, y)\,\mathsf{in}\, M$. Then, the initial answer type A^{ini} and final answer type A^{fin} of $\sigma(V; y.\, \mathsf{return}\, y)$ should be matched with those assigned to σ by operation signature Σ. Because the typing rule assumes $\Gamma, x : T^{\mathrm{ari}} \vdash M : \Sigma \triangleright T \,/\, A \Rightarrow A^{\mathrm{ini}}$, the initial and final answer types of the let expression $\mathsf{let}\, x = \sigma(V; y.\, \mathsf{return}\, y)\,\mathsf{in}\, M$ can be the types A and A^{fin}, respectively, by following (HT_Let). The rule (HT_Handle) typechecks a handling construct with H handle M if: the operation signature Σ used to typecheck the handled term M is matched with the operation clauses in the effect handler H; and the initial answer type of M is matched with the type of the return clause's body (this is required because the delimited continuation of the

[5] As the pure control effect $\Box$ only allows installing an unbounded number of effect handlers on *pure* terms, the encoding of natural numbers might be still rejected even in the presence of $\Box$. We leave this question open for future work.

handled term M is with H handle return $[\,]$, whose behavior is determined by the return clause). Then, (HT_Handle) assigns to the handling construct the final answer type C^{fin} of the handled term because it specifies the term to which the handling construct is finally transformed.

4.4 Examples

This section shows that HEPCF$^{\mathsf{ATM}}$ is expressive enough to accept effect handlers for exception handling, mutable state, and nondeterminism. We also present an example that cannot be typechecked in HEPCF$^{\mathsf{ATM}}$ due to the restriction by ATM. Finally, we compare HEPCF$^{\mathsf{ATM}}$ with the calculus GEPCF [Dal Lago and Ghyselen 2024] from the perspective of expressivity via the examples.

Well-Typed Examples. We first show three effect handlers that are successfully typechecked in HEPCF$^{\mathsf{ATM}}$.

Example 4.1 (Exception Handling). Let $H_{\mathsf{R}} \stackrel{\text{def}}{=} \{\mathsf{return}\, x \mapsto \mathsf{return}\, x\} \uplus \{\mathbf{Raise}(x; k) \mapsto \mathsf{return}\, \underline{1}\}$. This effect handler provides a clause of the operation **Raise** that discards the parameter x and the continuation k and simply returns the enum constant $\underline{1}$. Thus, for a term with H_{R} handle M, if the handled term M calls **Raise**, the handling construct returns $\underline{1}$. When M evaluates to a value, the handling construct returns the value itself as the return clause of H_{R} returns a given value x. An operation signature Σ_{R} for H_{R} can be given by

$$\{\mathbf{Raise} : \mathsf{unit} \rightsquigarrow 0 \,/\, C \Rightarrow (\Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A)\}$$

for arbitrary C, Σ, A, and $n \geq 1$. Then, the type system typechecks the term with H_{R} handle M as

$$\frac{\begin{array}{c}\Gamma \vdash M : \Sigma_{\mathsf{R}} \triangleright T \,/\, (\Sigma' \triangleright T \,/\, A' \Rightarrow A') \Rightarrow C^{\mathsf{fin}} \\ \Gamma, x : T \vdash \mathsf{return}\, x : \Sigma' \triangleright T \,/\, A' \Rightarrow A' \\ \Gamma, x : \mathsf{unit}, k_i : 0 \to C \vdash \mathsf{return}\, \underline{1} : \Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A\end{array}}{\Gamma \vdash \mathsf{with}\, H_{\mathsf{R}}\, \mathsf{handle}\, M : C^{\mathsf{fin}}}\ \text{HT\_Handle}$$

where Σ', T, A', and C^{fin} can be arbitrary as long as the first premise is derivable. If the term M may call **Raise** but does not modify answer types, we can let $T = \mathsf{n}$, $\Sigma' = \Sigma$, $A' = A$, and $C = C^{\mathsf{fin}} = \Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A$ and simplify the typing derivation as:

$$\frac{\begin{array}{c}\Gamma \vdash M : \Sigma_{\mathsf{R}} \triangleright \mathsf{n} \,/\, (\Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A) \Rightarrow (\Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A) \\ \Gamma, x : \mathsf{n} \vdash \mathsf{return}\, x : \Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A \\ \Gamma, x : \mathsf{unit}, k_i : 0 \to C \vdash \mathsf{return}\, \underline{1} : \Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A\end{array}}{\Gamma \vdash \mathsf{with}\, H_{\mathsf{R}}\, \mathsf{handle}\, M : (\Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A)}\ \text{HT\_Handle}$$

Example 4.2 (Mutable State). Consider the following effect handler H_{SG} for the operations **Set** and **Get** to manipulate a global Boolean reference:

$$\begin{array}{rl} H_{\mathsf{SG}} \stackrel{\text{def}}{=} & \{\mathsf{return}\, x \mapsto \mathsf{return}\, \lambda y.\mathsf{return}\, x\} \uplus \\ & \{\mathbf{Set}(x; k) \mapsto \mathsf{return}\, \lambda y.\mathsf{let}\, z = k\, \underline{1}\, \mathsf{in}\, z\, x\} \uplus \{\mathbf{Get}(x; k) \mapsto \mathsf{return}\, \lambda y.\mathsf{let}\, z = k\, y\, \mathsf{in}\, z\, y\}\,. \end{array}$$

Let $C \stackrel{\text{def}}{=} \Sigma \triangleright (2 \to \Sigma \triangleright T \,/\, A \Rightarrow A) \,/\, A \Rightarrow A$ for some Σ, T, and A. If we admit identifying the base type bool with the enum type $\underline{2}$, the effect handler H_{SG} can be typechecked against the operation signature Σ_{SG} defined to be

$$\{\mathbf{Set} : \mathsf{bool} \rightsquigarrow 1 \,/\, C \Rightarrow C\} \uplus \{\mathbf{Get} : \mathsf{unit} \rightsquigarrow 2 \,/\, C \Rightarrow C\}\,.$$

For example, **Set**'s clause is typechecked as follows:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma' \vdash k\,\underline{1} : C}\,\text{HT\_App} \quad \dfrac{\vdots}{\Gamma', z : 2 \to \Sigma \triangleright T \,/\, A \Rightarrow A \vdash z\,x : \Sigma \triangleright T \,/\, A \Rightarrow A}\,\text{HT\_App}}{\Gamma' \vdash \mathsf{let}\, z = k\,\underline{1}\,\mathsf{in}\, z\,x : \Sigma \triangleright T \,/\, A \Rightarrow A}\,\text{HT\_Let}}{\Gamma, x : \mathsf{bool}, k : 1 \to C \vdash \mathsf{return}\, \lambda y.\mathsf{let}\, z = k\,\underline{1}\,\mathsf{in}\, z\,x : C}\,\text{HT\_Return, HT\_Abs}$$

where $\Gamma' = \Gamma, x : \mathsf{bool}, k : 1 \to C, y : 2$. We can also typecheck **Get**'s clause in a similar way.

Example 4.3 (Nondeterminism). As the third example, we consider nondeterministic computation with an effect operation **Decide**. The operation **Decide** returns a value of the enum type 2, so its continuation may have two branches and the return value of **Decide** decides which branch is executed. We define an operation signature $\Sigma_{\mathbf{D}}$ for **Decide** to be

$$\{\mathbf{Decide} : \mathsf{unit} \rightsquigarrow 2 \,/\, (\Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A) \Rightarrow (\Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A)\}$$

for some Σ, A, and n, where the type n is of the values returned by the branches of **Decide**'s caller. As an implementation of **Decide**, we give the following effect handler:

$$H_{\mathbf{D}} \stackrel{\text{def}}{=} \{\mathsf{return}\, x \mapsto \mathsf{return}\, x\} \uplus \{\mathbf{Decide}(x; k) \mapsto \mathsf{let}\, y_1 = k\,\underline{1}\,\mathsf{in}\,\mathsf{let}\, y_2 = k\,\underline{2}\,\mathsf{in}\,\mathsf{max}(y_1, y_2)\}$$

where $\mathsf{max}(y_1, y_2)$ is a term returning the lager enum constant between y_1 and y_2. A handling construct installing $H_{\mathbf{D}}$ returns the maximum value among the results of the branches of the handled term. The effect handler $H_{\mathbf{D}}$ is typechecked against $\Sigma_{\mathbf{D}}$ as follows:

$$\dfrac{\Gamma \vdash k\,\underline{1} : \Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A \qquad \Gamma, y_1 : \mathsf{n} \vdash k\,\underline{2} : \Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A \quad \Gamma, y_1 : \mathsf{n}, y_2 : \mathsf{n} \vdash \mathsf{max}(y_1, y_2) : \Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A}{\Gamma \vdash \mathsf{let}\, y_1 = k\,\underline{1}\,\mathsf{in}\,\mathsf{let}\, y_2 = k\,\underline{2}\,\mathsf{in}\,\mathsf{max}(y_1, y_2) : \Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A}\,\text{HT\_Let}$$

where $\Gamma = \Gamma', x : \mathsf{unit}, k : 2 \to \Sigma \triangleright \mathsf{n} \,/\, A \Rightarrow A$ for some Γ'.

Ill-Typed Examples. As explained in Section 2, ATM bounds the number of effect handlers made active at the same time. More concretely, for an $\mathsf{HEPCF}^{\mathsf{ATM}}$ term M of a computation type $\Sigma_1 \triangleright T_1 \,/\, C_1 \Rightarrow (\Sigma_2 \triangleright T_2 \,/\, C_2 \Rightarrow (\cdots(\Sigma_n \triangleright T_n \,/\, C_n \Rightarrow T)\cdots))$, our type system only allows installing at most $n - 1$ effect handlers on the term M. Therefore, for instance, a term with H_n handle (with H_{n-1} handle $(\cdots$ (with H_1 handle $M)$ $\cdots))$ with n effect handlers $H_1, \cdots, H_n$ is ill typed. If it is well typed, the type of the term with H_{n-1} handle $(\cdots$ (with H_1 handle $M)$ $\cdots)$ with the $n - 1$ effect handlers should be $\Sigma_n \triangleright T_n \,/\, C_n \Rightarrow T$, but our type system does not allow handling a term of the type $\Sigma_n \triangleright T_n \,/\, C_n \Rightarrow T$ because it requires the answer types of a handled term to be computation types. This restriction of ATM on active effect handlers leads to the rejection of the term that Dal Lago and Ghyselen [2024] provide to show that algebraic effects and handlers can encode natural numbers.

To take a closer look at the restriction, we present another term that requires an unbounded number of effect handlers activated at the same time for being typechecked. It is a variant of the program given by Kawamata et al. [2024].

Example 4.4 (Unbounded Number of Active Effect Handlers). Consider a recursive function $V \stackrel{\text{def}}{=} \mathsf{fix}\, f.\lambda x.\mathsf{with}\, H\, \mathsf{handle}\, (f\, x)$ with an effect handler $H = \{\mathsf{return}\, x \mapsto \mathsf{return}\, x\}$. Applying this function causes an unbounded number of effect handlers to be active at the same time:

$$V\,() \longrightarrow^* \mathsf{with}\, H\, \mathsf{handle}\, (V\,()) \longrightarrow^* \mathsf{with}\, H\, \mathsf{handle}\, (\mathsf{with}\, H\, \mathsf{handle}\, (V\,())) \longrightarrow^* \cdots$$

Thus, it invalidates the restriction imposed by ATM.

To see that the function V is ill typed in HEPCF$^{\mathsf{ATM}}$ in fact, assume that it can be given a type $\mathsf{unit} \to \emptyset \triangleright \mathsf{unit} \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}}$ for some answer types A^{ini} and A^{fin}. Then, the typing judgment

$$f : \mathsf{unit} \to \emptyset \triangleright \mathsf{unit} \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}}, x : \mathsf{unit} \vdash \mathsf{with}\, H\, \mathsf{handle}\, (f\, x) : \emptyset \triangleright \mathsf{unit} \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}}$$

should be derivable. As only (HT_Handle) can be applied to derive it, we can find for the handled term $f\,x$, the typing judgment

$$f : \mathsf{unit} \to \emptyset \triangleright \mathsf{unit} \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}}, x : \mathsf{unit} \vdash f\, x : \emptyset \triangleright T \,/\, C \Rightarrow (\emptyset \triangleright \mathsf{unit} \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}})$$

holds for some T and C. Because the return type of f should be matched with the computation type of the application $f\,x$, the equations $T = \mathsf{unit}$, $C = A^{\mathrm{ini}}$, and $\emptyset \triangleright \mathsf{unit} \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}} = A^{\mathrm{fin}}$ have to hold. However, the last equation does not hold actually due to the circularity of A^{fin}.

Comparison with GEPCF. The calculus GEPCF proposed by Dal Lago and Ghyselen [2024] is a variant of HEPCF and restricts the operation clauses of effect handlers to be the form $\sigma(x; k) \mapsto \mathsf{let}\, y = M\, \mathsf{in}\, k\, y$ where $k \notin fv(M)$. This restriction of GEPCF enables the guarantee for the decidability of the HOMC problem. However, it causes the rejection of the aforementioned effect handlers H_{R}, H_{SG}, and H_{D} since **Raise**' clause in H_{R} discards a given continuation, **Set** and **Get**'s clauses in H_{SG} apply the continuation in a lambda abstraction, and **Decide**'s clause in H_{D} applies the continuation twice. On the other hand, the function in Example 4.4 would be well typed in GEPCF because GEPCF does not restrict the number of effect handlers that can be activated at the same time. As a result, the expressivity of HEPCF$^{\mathsf{ATM}}$ and that of GEPCF are incomparable.

4.5 Basic Properties

The calculus HEPCF$^{\mathsf{ATM}}$ also satisfies progress, subject reduction, and determinacy as EPCF.

Lemma 5 (Progress). *If $\emptyset \vdash M : C$, then one of the following holds: $M = \mathsf{return}\, V$ for some V; $M = \sigma(V; x.\, M')$ for some σ, V, x, and M'; or $M \longrightarrow M'$ for some M'.*

Lemma 6 (Subject Reduction). *If $\Gamma \vdash M : C$ and $M \longrightarrow M'$, then $\Gamma \vdash M' : C$.*

Lemma 7 (Determinacy). *If $M \longrightarrow M_1$ and $M \longrightarrow M_2$, then $M_1 = M_2$.*

5 Model Checking of HEPCF$^{\mathsf{ATM}}$

This section formalizes HOMC for HEPCF$^{\mathsf{ATM}}$, gives its example, and shows the decidability of the HOMC problem via a CPS transformation from HEPCF$^{\mathsf{ATM}}$ to EPCF.

5.1 Definition

We first introduce the effect trees of HEPCF$^{\mathsf{ATM}}$ terms, which are defined as in EPCF. A type T is *ground* if it is a base type or an enum type.

Definition 8 (Effect Trees for HEPCF$^{\mathsf{ATM}}$ Computations). *Given an operation signature Σ and a type T, the tree constructor signature S_T^{Σ} is defined as follows:*

$$S_T^{\Sigma} \stackrel{\mathrm{def}}{=} \{\sigma : n+1 \mid \sigma : B \rightsquigarrow \mathsf{n} \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}} \in \Sigma\} \cup \{\mathsf{return}\, V : 0 \mid \emptyset \vdash V : T\} \cup \bigcup_c \{c : 0\}\,.$$

Given a term M such that $\emptyset \vdash M : \Sigma \triangleright T \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}}$, the effect tree of M, denoted by $\mathbf{ET}(M)$, is a tree in $\mathbf{Tree}_{S_T^{\Sigma}}$ defined by the following (possibly infinite) process:

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

Definition 9 (Top-Level Operation Signatures). *An operation signature* Σ *is* top-level *if, for any* $\sigma : T^{\mathrm{par}} \rightsquigarrow T^{\mathrm{ari}} \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}} \in \Sigma$, $T^{\mathrm{par}} = B$, $T^{\mathrm{ari}} = E$, *and* $A^{\mathrm{ini}} = A^{\mathrm{fin}} = T$ *for some* B, E, *and* T.

We consider that well-typed programs to be executed are terms of a computation type $\Sigma \triangleright T \,/\, T \Rightarrow T'$ where the return type and initial answer type are required to coincide. This requirement enables starting the execution of the programs by passing an identity function as an initial continuation.

Definition 10 (Higher-Order Model Checking Problem for $\mathsf{HEPCF}^{\mathsf{ATM}}$). *Given an APT and an* $\mathsf{HEPCF}^{\mathsf{ATM}}$ *term* M *such that* $\emptyset \vdash M : \Sigma \triangleright T \,/\, T \Rightarrow T'$ *for some top-level operation signature* Σ *and ground types* T *and* T', *is* $\mathbf{ET}(M)$ *accepted by the APT?*

5.2 Example

As an instance of HOMC for $\mathsf{HEPCF}^{\mathsf{ATM}}$, we consider verifying a term that invokes mutable state implemented by an effect handler and file manipulation implemented as a primitive effect.

Example 5.1. For readability, we write $\mathsf{let}\, x = \sigma(V)\, \mathsf{in}\, M$ for the term $\sigma(V; x.\, M)$, and $M_1; M_2$ for the term $\mathsf{let}\, x = M_1\, \mathsf{in}\, M_2$ with some fresh variable x. Let

$$M \stackrel{\mathrm{def}}{=} \mathsf{with}\, H_{\mathsf{SG}}\, \mathsf{handle}\, \mathsf{let}\, x = \mathbf{Get}(())\, \mathsf{in}\, \mathsf{let}\, x' = \mathsf{case}(x; \mathsf{return}\, \mathsf{false}, \mathsf{return}\, \mathsf{true})\, \mathsf{in}\\ \mathbf{Set}(x'); \mathsf{let}\, z = \mathbf{Get}(())\, \mathsf{in}\, \mathsf{return}\, z\ .$$

The term M handles mutable Boolean state using the effect handler H_{SG} defined in Example 4.2. Consider a term

$$M_0 \stackrel{\mathrm{def}}{=} (\mathsf{fix}\, f.\lambda x.\mathbf{Open}(()); \mathsf{let}\, y = \mathbf{Read}(())\, \mathsf{in}\, \mathsf{let}\, z = M\, \mathsf{in}\, z\, y; \mathbf{Close}(()); f\, x)\ ()\ ,$$

which repeats executing the handled term in M with the contents y of the opened file as an initial state. In this term, the operations **Set** and **Get** are handled but **Open**, **Read**, and **Close** are unhandled. Therefore, only the latter operations appear in the effect tree generated by the term M_0. We can specify the semantics of the unhandled file operations and their desired property by the APT $\mathcal{A}_{\mathbf{File}}$ defined in Example 3.2.

5.3 CPS Transformation

We use the following shorthand to make the definition of our CPS transformation readable.

- A sequence of entities $a_1, \cdots, a_n$ is abbreviated to $\overline{a}$, and its length is denoted by $|\overline{a}|$. Given $\overline{a}$, we write a_i to designate the i-th element of the sequence $\overline{a}$.
- Given $\overline{x} = x_1, \cdots, x_n$, we write $\lambda\overline{x}.e$ for the function $\lambda x_1.\mathsf{return}\, \lambda x_2. \cdots\, \mathsf{return}\, \lambda x_n.e$.
- Given a term e and a sequence of values $\overline{v} = v_1, \cdots, v_n$ $(n > 0)$, we write $e\,\overline{v}$ for the EPCF term $\mathsf{let}\, x_0 = e\, \mathsf{in}\, \mathsf{let}\, x_1 = x_0\, v_1\, \mathsf{in}\, \mathsf{let}\, x_2 = x_1\, v_2\, \mathsf{in}\, \cdots\, \mathsf{let}\, x_{n-1} = x_{n-2}\, v_{n-1}\, \mathsf{in}\, x_{n-1}\, v_n$ where the variables $x_0, x_1, \cdots x_{n-1}$ are assumed to be fresh.
- For a computation type $C = \Sigma \triangleright T \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}}$, we write $C.\Sigma$ to designate the operation signature Σ.

We also assume that the set of all the operations is totally ordered to ensure the uniqueness of the CPS transformation.

We now define a CPS transformation for types, values, and terms in $\mathsf{HEPCF}^{\mathsf{ATM}}$, which will be explained in detail shortly. Our CPS transformation is a variant of the CPS transformation given by Kawamata et al. [2024], but they are different in two points. First, our CPS transformation does not incur *administrative reduction* (i.e., reduction that does not correspond to any reduction in a source term) as Plotkin's colon translation [Plotkin 1975]. Second, it does not require polymorphism, while Kawamata et al.'s CPS transformation requires it to address the pure control effect $\Box$. Our CPS transformation is defined as if it were to transform values and terms, but actually it is defined on

$\boxed{[\![T]\!]}$ **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**

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

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

$$[\![\emptyset]\!][\,\tau\,] \stackrel{\text{def}}{=} \tau$$
$$[\![\Sigma \uplus \{\sigma : T^{\text{par}} \rightsquigarrow T^{\text{ari}} \,/\, A^{\text{ini}} \Rightarrow A^{\text{fin}}\}]\!][\,\tau\,] \stackrel{\text{def}}{=} ([\![T^{\text{par}}]\!] \to ([\![T^{\text{ari}}]\!] \to [\![A^{\text{ini}}]\!]) \to [\![A^{\text{fin}}]\!]) \to [\![\Sigma]\!][\,\tau\,]$$
(where σ is lower than any operation in Σ)

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

$$[\![x]\!] \stackrel{\text{def}}{=} x \qquad [\![c]\!] \stackrel{\text{def}}{=} c \qquad [\![\underline{n}]\!] \stackrel{\text{def}}{=} \underline{n} \qquad [\![\lambda x.M]\!] \stackrel{\text{def}}{=} \lambda x.\mathsf{return}\,[\![M]\!] \qquad [\![\mathsf{fix}\,x.V]\!] \stackrel{\text{def}}{=} \mathsf{fix}\,x.[\![V]\!]$$

$\boxed{[\![M]\!]}$ **for thunks**

$$[\![M]\!] \stackrel{\text{def}}{=} \lambda \overline{h}, k.[\![M]\!][\,\overline{h} \mid k\,] \quad \text{(where } \overline{h} \text{ and } k \text{ are fresh and } |\overline{h}| = |\Sigma|\text{)}$$

$\boxed{[\![M]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,]}$ **for terms with operation clauses and continuations** ($|\overline{v^{\mathsf{h}}}| = |\Sigma|$ is assumed)

$$[\![\mathsf{return}\,V]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,] \stackrel{\text{def}}{=} v^{\mathsf{k}}\,[\![V]\!]$$
$$[\![\mathsf{let}\,x = M_1\,\mathsf{in}\,M_2]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,] \stackrel{\text{def}}{=} [\![M_1]\!][\,\overline{v^{\mathsf{h}}} \mid \lambda x.[\![M_2]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,]\,]$$
$$[\![V_1\,V_2]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,] \stackrel{\text{def}}{=} [\![V_1]\!]\,[\![V_2]\!]\,\overline{v^{\mathsf{h}}}\,v^{\mathsf{k}}$$
$$[\![\mathsf{case}(V; M_1, \cdots, M_n)]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,] \stackrel{\text{def}}{=} \mathsf{case}([\![V]\!]; [\![M_1]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,], \cdots, [\![M_n]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,])$$
$$[\![\sigma_i(V; x.\,M)]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,] \stackrel{\text{def}}{=} v^{\mathsf{h}}_i\,[\![V]\!]\,\lambda x, \overline{h}, k.[\![M]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,]\,\overline{h}\,k$$
(if σ_i is the i-th operation in Σ and $\sigma_i : T^{\text{par}}_i \rightsquigarrow T^{\text{ari}}_i \,/\, C^{\text{ini}}_i \Rightarrow A^{\text{fin}}_i \in \Sigma$ and $|C^{\text{ini}}_i.\Sigma| = |\overline{h}|$ and $\overline{h}$ and k are fresh)

$$[\![\sigma_i(V; x.\,M)]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,] \stackrel{\text{def}}{=} v^{\mathsf{h}}_i\,[\![V]\!]\,\lambda x.[\![M]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,]$$
(if σ_i is the i-th operation in Σ and $\sigma_i : T^{\text{par}}_i \rightsquigarrow T^{\text{ari}}_i \,/\, T^{\text{ini}}_i \Rightarrow A^{\text{fin}}_i \in \Sigma$)

$$[\![\mathsf{with}\,H\,\mathsf{handle}\,M]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,] \stackrel{\text{def}}{=} [\![M]\!][\,V_1, \cdots, V_n \mid \lambda x.\mathsf{return}\,[\![M_0]\!]\,]\,\overline{v^{\mathsf{h}}}\,v^{\mathsf{k}}$$
(where $H = \{\mathsf{return}\,x \mapsto M_0\} \uplus \{\sigma_i(x_i; k_i) \mapsto M_i\}^{1 \le i \le n}$ and $\forall\, i \in [1, n].\ V_i = \lambda x_i, k_i.\mathsf{return}\,[\![M_i]\!]$)

Fig. 7. CPS transformation. In the definition of $[\![M]\!]$ and $[\![M]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,]$, we assume that the computation type of M is composed of an operation signature Σ.

their typing derivations. Thus, we assume that a computation type—more specifically, its operation signature—used to typecheck terms is given and mention it in defining the CPS transformation.

Definition 11 (CPS Transformation of Types, Values, and Terms). *CPS Transformation* $[\![-]\!]$ *from* $\mathsf{HEPCF}^{\mathsf{ATM}}$ *to* EPCF *is defined in Figure 7, mapping*

- *value types* T *to* EPCF *types* $[\![T]\!]$,
- *computation types* C *to* EPCF *types* $[\![C]\!]$,
- *operation signatures* Σ *to functions that, given a* EPCF *type* τ*, return the* EPCF *type* $[\![\Sigma]\!][\,\tau\,]$,
- *values* V *to* EPCF *values* $[\![V]\!]$,
- *terms* M *to* EPCF *values* $[\![M]\!]$*, and*
- *terms* M *to* EPCF *terms* $[\![M]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,]$ *given values* $\overline{v^{\mathsf{h}}}$ *and* v^{k}.

We also write $[\![\Gamma]\!]$ *for the* EPCF *typing context obtained by CPS-transforming the types of all the bindings of a typing context* Γ.

CPS Transformation of Types. The CPS transformation is defined for both value and computation types. The transformation of value types is straightforward. The transformation of a computation type $C \stackrel{\text{def}}{=} \Sigma \triangleright T \,/\, A^{\text{ini}} \Rightarrow A^{\text{fin}}$ passes $([\![T]\!] \to [\![A^{\text{ini}}]\!]) \to [\![A^{\text{fin}}]\!]$ to the function $[\![\Sigma]\!][-]$. The type $([\![T]\!] \to [\![A^{\text{ini}}]\!]) \to [\![A^{\text{fin}}]\!]$ indicates that how the answer type modification is expressed in CPS-transformed terms: an $\mathsf{HEPCF}^{\mathsf{ATM}}$ term of the type C is transformed into an EPCF term that takes the continuations of the type $[\![T]\!] \to [\![A^{\text{ini}}]\!]$ and returns an "answer" value of the type $[\![A^{\text{fin}}]\!]$. The function $[\![\Sigma]\!][-]$ wraps the passed type to take the CPS-transformed operation clauses of an effect handler matched with the signature Σ. It signifies that an CPS-transformed operation clause takes a parameter and a continuation.

The CPS transformation reveals that computation types in $\mathsf{HEPCF}^{\mathsf{ATM}}$ bound the number of continuations passed to $\mathsf{HEPCF}^{\mathsf{ATM}}$ terms. In general, a computation type takes the form $\Sigma_1 \triangleright T_1 \,/\, A_1 \Rightarrow (\Sigma_2 \triangleright T_2 \,/\, A_2 \Rightarrow (\cdots (\Sigma_n \triangleright T_n \,/\, A_n \Rightarrow T) \cdots))$. Let us ignore the operation signatures $\Sigma_1, \cdots, \Sigma_n$ now as they are not important here. Then, the computation type is CPS-transformed into the type $([\![T_1]\!] \to [\![A_1]\!]) \to ([\![T_2]\!] \to [\![A_2]\!]) \to \cdots \to ([\![T_n]\!] \to [\![A_n]\!]) \to [\![T]\!]$, which ensures that a CPS-transformed term of this type takes only n continuations. This "bounded finiteness" for the number of continuations is critical to guarantee the decidability of the HOMC problem. The use of an unbounded number of continuations allows encoding arbitrary data of an infinite domain, such as natural numbers, into the control flow, as shown by Dal Lago and Ghyselen [2024] for HEPCF which does not bound the number of continuations passed to effectful terms.

CPS Transformation of Values and Terms. The CPS transformation for values is defined straightforwardly. For terms, there are two kind of transformations. One is $[\![M]\!]$, which transforms a "thunk" term M, which corresponds to the body of a function, a return clause, or an operation clause. The thunks expect operation clauses and a continuation to be passed from a call site. Another is $[\![M]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,]$, which transforms a term M with CPS-transformed operation clauses $\overline{v^{\mathsf{h}}}$ and a continuation v^{k}. Kawamata et al.'s CPS transformation takes them as object-level arguments, but it causes administrative reduction and prevents us from proving that the evaluation of a source term is simulated only by the evaluation sequence of its CPS-transformed result (Lemma 10). Passing the operation clauses and the continuation during the transformation enables us to get rid of administrative reduction, which leads to establishing a more precise semantic relationship between source and CPS-transformed terms. If we only focus on the decidability of HOMC, we could admit administrate reduction. However, we expect that identifying the precise semantic relationship between source and CPS-transformed terms would be useful in studying the time complexity of HOMC via the CPS transformation, although the research on time complexity is left as future work.

Finally, it should be noted how we transform an algebraic operation call $\sigma_i(V; x.\, M)$, which depends on what initial answer type is assigned to σ_i. If it is a computation type, the operation call may be handled by an effect handler. Then, the continuation passed to the CPS-transformed operation clause v_i^{h} of σ_i should be functionalized because v_i^{h} uses the continuation as a function. As we mentioned, the body of a function is a thunk, which takes operation clauses and continuations

from call sites. Thus, in this case, the continuation is eta-expanded to take them as $\overline{h}$ and k (if the continuation is not eta-expanded, an administrative reduction corresponding to the eta-expansion will happen). If the initial answer type of σ_i is a value type T_i^{ini}, then the eta-expanded continuation cannot be typed at $[\![T_i^{\mathrm{ari}}]\!] \to [\![T_i^{\mathrm{ini}}]\!]$, where T_i^{ari} is the arity type of σ. This would result in breaking type preservation of the CPS transformation. Therefore, in this case, we use the continuation not being eta-expanded.

5.4 Properties

First, we show that the effect tree of a well-typed $\mathsf{HEPCF}^{\mathsf{ATM}}$ term is well defined.

Lemma 8 (Well-Definedness of $\mathsf{HEPCF}^{\mathsf{ATM}}$ Effect Trees). *If* $\emptyset \vdash M : \Sigma \triangleright T \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}}$ *and* Σ *is top-level, then* $\mathbf{ET}(M)$ *is well defined and uniquely determined, and it is in* $\mathbf{Tree}_{S_T^\Sigma}$.

In the rest, we show the properties of the CPS transformation and how it enables to reduce the decidability of HOMC for $\mathsf{HEPCF}^{\mathsf{ATM}}$ to that for EPCF.

We start by showing that our CPS transformation is type- and semantics-preserving. We denote the set of the variables in a typing context Δ by $dom(\Delta)$, and write $\Delta_1 \preceq \Delta_2$ if $dom(\Delta_1) \subseteq dom(\Delta_2)$ and, for any $x \in dom(\Delta_1)$, $\Delta_1(x) = \Delta_2(x)$. We also write $e \longrightarrow^+ e'$ if $e \longrightarrow^n e'$ for some $n > 0$.

Definition 12 (Typed CPS Operation Clauses). *Let* $\Sigma = \{\sigma_i : T_i^{\mathrm{par}} \rightsquigarrow T_i^{\mathrm{ari}} \,/\, A_i^{\mathrm{ini}} \Rightarrow A_i^{\mathrm{fin}}\}^{1 \le i \le n}$ *where* $\sigma_1, \cdots, \sigma_n$ *are ordered. For a value sequence* $\overline{v^{\mathsf{h}}} = v_1^{\mathsf{h}}, \cdots, v_n^{\mathsf{h}}$, *we write* $\Xi \parallel \Delta \vdash \overline{v^{\mathsf{h}}} : \Sigma$ *if, for each* $i \in [1, n]$, $\Xi \parallel \Delta \vdash v_i^{\mathsf{h}} : [\![T_i^{\mathrm{par}}]\!] \to ([\![T_i^{\mathrm{ari}}]\!] \to [\![A_i^{\mathrm{ini}}]\!]) \to [\![A_i^{\mathrm{fin}}]\!]$ *holds.*

Lemma 9 (Type Preservation of the CPS Transformation). *Assume that* $[\![\Gamma]\!] \preceq \Delta$.

- *If* $\Gamma \vdash V : T$, *then* $\Xi \parallel \Delta \vdash [\![V]\!] : [\![T]\!]$ *for any* Ξ.
- *If* $\Gamma \vdash M : \Sigma \triangleright T \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}}$ *and* $\Xi \parallel \Delta \vdash \overline{v^{\mathsf{h}}} : \Sigma$ *and* $\Xi \parallel \Delta \vdash v^{\mathsf{k}} : [\![T]\!] \to [\![A^{\mathrm{ini}}]\!]$, *then* $\Xi \parallel \Delta \vdash [\![M]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,] : [\![A^{\mathrm{fin}}]\!]$.
- *If* $\Gamma \vdash M : C$, *then* $\Xi \parallel \Delta \vdash [\![M]\!] : [\![C]\!]$ *for any* Ξ.

Proof. Straightforward by mutual induction on the typing derivations. □

Lemma 10 (Simulation up to Reduction). *If* $\Gamma \vdash M : \Sigma \triangleright T \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}}$ *and* $M \longrightarrow M'$, *then, for any* $\overline{v^{\mathsf{h}}}$ *and* v^{k} *such that* $|\overline{v^{\mathsf{h}}}| = |\Sigma|$, *either of the following holds:*

- $M' \longrightarrow^* \sigma(V_0; x.\, M_0)$ *and* $[\![M]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,] = [\![\sigma(V_0; x.\, M_0)]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,]$ *for some* σ, V_0, x, M_0; *or*
- $M' \longrightarrow^* M''$ *and* $[\![M]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,] \longrightarrow^+ [\![M'']\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,]$ *for some* M''.

Proof. The proof is straightforward by case analysis on the evaluation $M \longrightarrow M'$. □

Next, we show that the CPS transformation is "compatible" with effect tree generation. Before that, we define CPS transformation for $\mathsf{HEPCF}^{\mathsf{ATM}}$ effect trees.

Definition 13 (CPS Transformation for Effect Trees). *Given an* $\mathsf{HEPCF}^{\mathsf{ATM}}$ *effect tree* $\mathbf{ET}(M)$ *and an* EPCF *value* v, *the tree* $[\![\mathbf{ET}(M)]\!][\,v\,]$ *is defined coinductively as follows:*

$$[\![\bot]\!][\,v\,] \stackrel{\mathrm{def}}{=} \bot \qquad [\![\mathsf{return}\ V]\!][\,v\,] \stackrel{\mathrm{def}}{=} \mathbf{ET}(v\,[\![V]\!])$$
$$[\![\sigma(c, \mathbf{ET}(M_1), \cdots, \mathbf{ET}(M_n))]\!][\,v\,] \stackrel{\mathrm{def}}{=} \sigma(c, [\![\mathbf{ET}(M_1)]\!][\,v\,], \cdots, [\![\mathbf{ET}(M_n)]\!][\,v\,])$$

Lemma 11 (Compatibility between CPS Transformation and Effect Tree Generation). *Let* $\Sigma = \{\sigma_i : B_i \rightsquigarrow E_i \,/\, T_i \Rightarrow T_i\}^{1 \le i \le n}$ *and* $\Xi = \{\sigma_i : B_i \rightsquigarrow E_i\}^{1 \le i \le n}$. *Assume that* $\emptyset \vdash M : \Sigma \triangleright T \,/\, A^{\mathrm{ini}} \Rightarrow A^{\mathrm{fin}}$ *and* $\sigma_1, \cdots, \sigma_n$ *are ordered. Let* $\overline{v^{\mathsf{h}}} = v_1^{\mathsf{h}}, \cdots, v_n^{\mathsf{h}}$ *such that, for any* $i \in [1, n]$, $v_i^{\mathsf{h}} = \lambda x, k.\, \sigma_i(x; y.\, k\, y)$ *for some distinct variables* x, k, *and* y. *Also, let* v^{k} *be a value such that* $\Xi \parallel \emptyset \vdash v^{\mathsf{k}} : [\![T]\!] \to [\![A^{\mathrm{ini}}]\!]$. *Then,* $\mathbf{ET}([\![M]\!][\,\overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\,]) = [\![\mathbf{ET}(M)]\!][\,v^{\mathsf{k}}\,]$.

Proof. By coinduction on the equivalence of trees. It is easy to confirm that both $[\![\mathbf{ET}(M)]\!][\ v^{\mathsf{k}}\ ]$ and $\mathbf{ET}([\![M]\!][\ \overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\ ])$ are in $\mathbf{Tree}_{S'}$ where $S' = S^{\Xi}_{[\![A^{\mathrm{fin}}]\!]}$. □

Lemma 11 indicates that, by taking the identity continuation as v^{k}—it is possible when $T = A^{\mathrm{ini}}$—and assuming the value type T of M to be ground—that is, to be a base type or an enum type—we can conclude $\mathbf{ET}([\![M]\!][\ \overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\ ]) = \mathbf{ET}(M)$.

Theorem 2 (Preservation of Effect Trees). *Let $\Sigma = \{\sigma_i : B_i \rightsquigarrow E_i \,/\, T_i \Rightarrow T_i\}^{1 \le i \le n}$ and T be a ground type. Assume that $\emptyset \vdash M : \Sigma \triangleright T \,/\, T \Rightarrow A^{\mathrm{fin}}$ and $\sigma_1, \cdots, \sigma_n$ are ordered. Let $\overline{v^{\mathsf{h}}} = v_1^{\mathsf{h}}, \cdots, v_n^{\mathsf{h}}$ such that, for any $i \in [1, n]$, $v_i^{\mathsf{h}} = \lambda x, k.\sigma_i(x; y.\, k\, y)$ for some distinct variables x, k, and y. Also, let $v^{\mathsf{k}} = \lambda x.\mathsf{return}\, x$. Then, $\mathbf{ET}([\![M]\!][\ \overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\ ]) = \mathbf{ET}(M)$.*

Because $[\![M]\!][\ \overline{v^{\mathsf{h}}} \mid v^{\mathsf{k}}\ ]$ is of an EPCF term of $[\![A^{\mathrm{fin}}]\!]$ by the type preservation of the CPS transformation (Lemma 9), the HOMC problem for $\mathsf{HEPCF}^{\mathsf{ATM}}$ is decidable if A^{fin} is a ground value type, by the decidability for EPCF terms (Theorem 1).

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

We have proved that, *once an* $\mathsf{HEPCF}^{\mathsf{ATM}}$ *term is typechecked*, its HOMC is decidable. The remaining issue is whether the type inference problem of $\mathsf{HEPCF}^{\mathsf{ATM}}$ is decidable. In this paper, we do not formally address the type inference of $\mathsf{HEPCF}^{\mathsf{ATM}}$, but we conjecture that it is decidable because we can implement it by adapting a sound, complete, and decidable type inference algorithm for the delimited control operator set shift0/reset0 with ATM [Materzok and Biernacki 2011], which has the expressivity similar to effect handlers [Forster et al. 2017].

6 Implementation

In this section, we describe our prototype implementation of HOMC for $\mathsf{HEPCF}^{\mathsf{ATM}}$. Our tool EffCaml takes as input an $\mathsf{HEPCF}^{\mathsf{ATM}}$ term written in a subset of OCaml 5 (it enables the reuse of the OCaml 5 syntax to write programs with effect handlers) and an *alternating tree automaton* (ATA) as a specification of the primitive effects and the property of interest. ATAs are the same as APTs where parity conditions always hold. ATAs can express safety properties but not liveness properties in general, while APTs can express both. Our tool treats only ATAs because our backend higher-order model checker HorSat2 [Broadbent and Kobayashi 2013; Kobayashi 2016] only supports ATAs. Given the input, our tool first checks if the input term is well typed. This checking is implemented as type inference, which infers the type of the input term (and those of its subterms) according to $\mathsf{HEPCF}^{\mathsf{ATM}}$'s type system; if the type inference fails, the tool rejects the input. The type inference is implemented by adapting the unification-based type inference algorithm proposed by Materzok and Biernacki [2011] for the delimited control operator set shift0/reset0. Our implementation of the type inference is similar to the one in the refinement type checker developed by Kawamata et al. [2024]. The only difference from Kawamata et al. is that we need to prevent the use of the pure control effect □, but it only brings a minor change in the algorithm. If the type checking succeeds, our tool then transforms the given $\mathsf{HEPCF}^{\mathsf{ATM}}$ term along with the inferred type information. The transformation is based on the CPS transformation presented in Section 5.3. While the presented CPS transformation outputs an EPCF term, the transformation implemented in our tool outputs a HORS to reuse HorSat2 as the backend. Note that the adaptation of the CPS transformation to HORS poses no challenge. Finally, the obtained HORS and the input ATA are fed to HorSat2, which returns whether the HORS satisfies the specification expressed by the ATA.

Table 1. Experiment results of our HOMC tool.

File name	Lines of code	Safe/Unsafe	Result correct?	Time (sec.)
`file.ml`	53	Safe	Yes	0.005
`file_unsafe.ml`	52	Unsafe	Yes	0.004
`immutable_false.ml`	42	Safe	Yes	0.004
`immutable_mutable_set_not_b_false.ml`	64	Safe	Yes	0.004
`immutable_set_not_b_false.ml`	44	Safe	Yes	0.004
`immutable_true.ml`	42	Safe	Yes	0.004
`mutable_false.ml`	43	Safe	Yes	0.004
`mutable_immutable_set_not_b_false.ml`	63	Unsafe	Yes	0.004
`mutable_set_not_b_false.ml`	45	Unsafe	Yes	0.005
`mutable_true.ml`	43	Safe	Yes	0.003

To test the correctness and performance of our tool, we conducted a preliminary experiment with the examples presented in the paper. The experiment is conducted on the machine with 12th Gen Intel(R) Core(TM) i7-1270P 2.20 GHz, 32 GB of memory. Table 1 displays the experimental results. The instances that the program should behave as specified by the ATA are marked "Safe" (otherwise, marked "Unsafe"). The first two instances (`file.ml` and `file_unsafe.ml`) verify if the file operations are used correctly, and the others verify if programs with mutable state do not perform **Raise**. Due to the limitation of the backend model checker HorSat2, we only verify safety properties. The result shows that our tool successfully verifies or falsifies all the instances in a reasonable time, but it would be because the programs and ATAs are small. It is left as future work to evaluate our tool for larger, complex instances in terms of both programs and specifications.

7 Related Work

The verification of effectful higher-order programs have gained a lot of attention thus far. One of the automated verification methodologies for higher-order programs is the HOMC for HORS [Kobayashi 2009; Ong 2006]. As explained in Section 1, several extensions to effectful programs has been proposed [Kobayashi 2009; Kobayashi et al. 2019; Sato et al. 2013], but Dal Lago and Ghyselen [2024] is the first who successfully extended HOMC to general algebraic effects. They also considered an extension to effect handlers, proved the undecidability of the HOMC problem for the extension, defined GEPCF, a variant of PCF that only supports restricted effect handlers, and showed the decidability of the HOMC problem for GEPCF. As discussed in Section 4.4, $\mathsf{HEPCF}^{\mathsf{ATM}}$ and GEPCF are incomparable, and it is left open whether we can take the best of both worlds.

Another major approach is to use program logics. Several researchers have proposed program logics to address a variety of computation effects in a unified manner using algebraic effects [Kidney et al. 2024; Matache and Staton 2019; Plotkin and Pretnar 2008] or to reason about programs with effect handlers [de Vilhena and Pottier 2021; Luksic and Pretnar 2020]. These works aim to construct deductive proof systems for effectful programs, while we focus on a model checking approach (after typechecking), which facilitates automated verification.

A research line orthogonal to HOMC is the application of effect systems to safety or temporal verification in the presence of certain control operators including effect handlers [Gordon 2020; Kawamata et al. 2024; Sekiyama and Unno 2023; Song et al. 2022; Swamy et al. 2013]. An advantage of employing effect systems is that they can work well even on infinite data domains. However, it is unclear how efficiently and automatically these approaches work for *temporal* verification, while several HOMC tools that can address temporal verification have been developed [Kobayashi

et al. 2011, 2010; Ramsay et al. 2014]. Swamy et al. [2013] developed F*, a proof-oriented effectful language, but it only supports safety properties, while HOMC can also address liveness properties. Kawamata et al. [2024] defined a refinement type system for effect handlers and developed a verification tool based on it, but they also focus only on safety properties. Song et al. [2022] studied temporal *safety* verification, but temporal liveness verification is out of their scope.

Schuster et al. [2020] studied a compilation technique for *lexically scoped effect handlers*, which are passed as arguments to operation calls to be handled, unlike *dynamic* effect handlers addressed in this work, which are sought on a run-time continuation stack when an algebraic operation is called. For efficient compilation, Schuster et al.'s type system tracks a finite list of answer types. Schuster et al. call the answer type list a *stack shape*, which implements simplified ATM. In Schuster et al.'s type system, a type of a computation takes the form $[T']_{\overline{T}}$ where T' is the value type and $\overline{T}$ is the stack shape of the computation. When $\overline{T} = T_1, \cdots, T_n$, the type $[T']_{\overline{T}}$ corresponds to the $\mathsf{HEPCF}^{\mathsf{ATM}}$ computation type $T' \,/\, A_1 \Rightarrow A_1$ (without operation signatures because Schuster et al.'s type system do not consider them) with answer types $A_1, \ldots, A_n$ defined as, for $i \in [1, n-1]$, $A_i \stackrel{\text{def}}{=} T_i \,/\, A_{i+1} \Rightarrow A_{i+1}$ and $A_n \stackrel{\text{def}}{=} T_n$. This view is also encouraged by Schuster et al.'s iterated CPS transformation, which is defined similarly to ours. Thus, adapting stack shapes to dynamic effect handlers could guarantee the decidability of HOMC. However, unlike stack shapes, the full ATM can also modify answer types. This distinctive feature of ATM enables verifying the temporal safety properties of *handled* operations as well. For instance, Kawamata et al. [2024] showed that ATM enables verifying the safe use of the file operations even if they are implemented by an effect handler. It seems hard to verify such a property of handled operations using stack shapes.

8 Conclusion

This paper studies the decidability of the model checking problem for higher-order programs with algebraic effects and handlers. Dal Lago and Ghyselen [2024] showed that it is undecidable in general. We analyzed the counterexample to the decidability given by Dal Lago and Ghyselen and found that one cause of the undecidability is to make an unbounded number of effect handlers active at the same time, which enables the encoding of natural numbers. To prevent it while allowing a wide range of effect implementations by effect handlers, we apply answer-type modification (ATM), which not only allows modification of answer types but also bounds the number of effect handlers made active at the same time. To show that ATM ensures the decidability of the HOMC problem, we define a variant $\mathsf{HEPCF}^{\mathsf{ATM}}$ of PCF with effect handlers and ATM and give a type- and semantics-preserving CPS transformation from $\mathsf{HEPCF}^{\mathsf{ATM}}$ to EPCF for which HOMC is decidable.

There are several directions for future work. One important step for practice is to combine the present work with the approaches to extending HOMC to infinite data domains [Kobayashi and Igarashi 2013; Kobayashi et al. 2011, 2010; Matsumoto et al. 2015; Ong and Ramsay 2011; Unno et al. 2010]. Another direction is to analyze the time complexity of the HOMC problem for $\mathsf{HEPCF}^{\mathsf{ATM}}$. Because the HOMC problem for $\mathsf{HEPCF}^{\mathsf{ATM}}$ is reduced to that for EPCF, we could start by exploring the time complexity of the latter. Finally, it is known that there exists some relationship between HORS and higher-order fixpoint logic (HFL) [Kobayashi et al. 2017]. It is an interesting question whether there is such a relationship between effectful terms (or effect trees) and HFL.

Acknowledgments

We thank the anonymous reviewers for their fruitful comments on the submission. We are also grateful to Ugo Dal Lago for the comments on the early draft. This work was partly supported by JSPS KAKENHI Grant Numbers JP22K17875, JP24H00699, JP20H04162, JP22H03564, JP22H03570, and JP20H05703, as well as JST CREST Grant Number JPMJCR21M3.

References

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT Press.

Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6538)*, Ranjit Jhala and David A. Schmidt (Eds.). Springer, 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

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. https://doi.org/10.4230/LIPICS.CSL.2013.129

Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. 1990. Symbolic Model Checking: 10^20 States and Beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*. IEEE Computer Society, 428–439. https://doi.org/10.1109/LICS.1990.113767

Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). 2018. *Handbook of Model Checking*. Springer. https://doi.org/10.1007/978-3-319-10575-8

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. https://doi.org/10.1007/978-3-031-21314-4_4

Youyou Cong, Chiaki Ishio, Kaho Honda, and Kenichi Asai. 2021. A Functional Abstraction of Typed Invocation Contexts. In *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference) (LIPIcs, Vol. 195)*, Naoki Kobayashi (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 12:1–12:18. https://doi.org/10.4230/LIPICS.FSCD.2021.12

Ugo Dal Lago. 2024. Private communication.

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

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

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

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *PACMPL* 1, ICFP (2017), 13:1–13:29. https://doi.org/10.1145/3110257

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. https://doi.org/10.4230/LIPIcs.ECOOP.2020.23

Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. 2008. Collapsible Pushdown Automata and Recursion Schemes. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. IEEE Computer Society, 452–461. https://doi.org/10.1109/LICS.2008.34

Chiaki Ishio and Kenichi Asai. 2022. Type System for Four Delimited Control Operators. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2022, Auckland, New Zealand, December 6-7, 2022*, Bernhard Scholz and Yukiyoshi Kameyama (Eds.). ACM, 45–58. https://doi.org/10.1145/3564719.3568691

Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *ACM Comput. Surv.* 41, 4 (2009), 21:1–21:54. https://doi.org/10.1145/1592434.1592438

Yukiyoshi Kameyama and Takuo Yonezawa. 2008. Typed Dynamic Control Operators for Delimited Continuations. In *Functional and Logic Programming, 9th International Symposium, FLOPS 2008*. 239–254. https://doi.org/10.1007/978-3-540-78969-7_18

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

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. https://doi.org/10.1145/3633280

Donnacha Oisín Kidney, Zhixuan Yang, and Nicolas Wu. 2024. Algebraic Effects Meet Hoare Logic in Cubical Agda. *Proc. ACM Program. Lang.* 8, POPL (2024), 1663–1695. https://doi.org/10.1145/3632898

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. https://doi.org/10.1145/1480881.1480933

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

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

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. https://doi.org/10.1109/LICS.2019.8785679

Naoki Kobayashi and Atsushi Igarashi. 2013. Model-Checking Higher-Order Programs with Recursive Types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 431–450. https://doi.org/10.1007/978-3-642-37036-6_24

Naoki Kobayashi, Étienne Lozes, and Florian Bruse. 2017. On the relationship between higher-order recursion schemes and higher-order fixpoint logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 246–259. https://doi.org/10.1145/3009837.3009854

Naoki Kobayashi and C.-H. Luke Ong. 2009a. 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, 11-14 August 2009, Los Angeles, CA, USA*. IEEE Computer Society, 179–188. https://doi.org/10.1109/LICS.2009.29

Naoki Kobayashi and C.-H. Luke Ong. 2009b. 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. https://doi.org/10.1109/LICS.2009.29

Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 222–233. https://doi.org/10.1145/1993498.1993525

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. https://doi.org/10.1145/1706299.1706355

Sam Lindley. 2014. Algebraic effects and effect handlers for idioms and arrows. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, José Pedro Magalhães and Tiark Rompf (Eds.). ACM, 47–58. https://doi.org/10.1145/2633628.2633636

Ziga Luksic and Matija Pretnar. 2020. Local algebraic effect theories. *J. Funct. Program.* 30 (2020), e13. https://doi.org/10.1017/S0956796819000212

Cristina Matache and Sam Staton. 2019. A Sound and Complete Logic for Algebraic Effects. In *Foundations of Software Science and Computation Structures - 22nd International Conference, FOSSACS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11425)*, Mikolaj Bojanczyk and Alex Simpson (Eds.). Springer, 382–399. https://doi.org/10.1007/978-3-030-17127-8_22

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. https://doi.org/10.1145/2034773.2034786

Yuma Matsumoto, Naoki Kobayashi, and Hiroshi Unno. 2015. Automata-Based Abstraction for Automated Verification of Higher-Order Tree-Processing Programs. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9458)*, Xinyu Feng and Sungwoo Park (Eds.). Springer, 295–312. https://doi.org/10.1007/978-3-319-26529-2_16

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. https://doi.org/10.1109/LICS.2006.38

C.-H. Luke Ong and Steven J. Ramsay. 2011. Verifying higher-order functional programs with pattern-matching algebraic data types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 587–598. https://doi.org/10.1145/1926385.1926453

Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.* 1, 2 (1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

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

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

Gordon D. Plotkin and Matija Pretnar. 2008. A Logic for Algebraic Effects. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. IEEE Computer Society, 118–129. https://doi.org/10.1109/LICS.2008.45

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. https://doi.org/10.1007/978-3-642-00590-9_7

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

Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 46–57. https://doi.org/10.1109/SFCS.1977.32

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. https://doi.org/10.1016/J.ENTCS.2015.12.003

Steven J. Ramsay, Robin P. Neatherway, and C.-H. Luke Ong. 2014. A type-directed abstraction refinement approach to higher-order model checking. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 61–72. https://doi.org/10.1145/2535838.2535873

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2 (ACM '72)*. 717–740. https://doi.org/10.1145/800194.805852

Sylvain Salvati and Igor Walukiewicz. 2014. Krivine machines and higher-order schemes. *Inf. Comput.* 239 (2014), 340–355. https://doi.org/10.1016/J.IC.2014.07.012

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. https://doi.org/10.1145/2426890.2426900

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling effect handlers in capability-passing style. *Proc. ACM Program. Lang.* 4, ICFP (2020), 93:1–93:28. https://doi.org/10.1145/3408975

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. https://doi.org/10.1145/3571264

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. https://doi.org/10.1007/978-3-031-21037-2_5

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the dijkstra monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 387–398. https://doi.org/10.1145/2491956.2491978

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. https://doi.org/10.1007/978-3-642-12032-9_24

Hiroshi Unno, Naoshi Tabuchi, and Naoki Kobayashi. 2010. Verification of Tree-Processing Programs via Higher-Order Model Checking. In *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6461)*, Kazunori Ueda (Ed.). Springer, 312–327. https://doi.org/10.1007/978-3-642-17164-2_22

Received 2024-04-06; accepted 2024-08-18