On Model-Checking Higher-Order Effectful Programs*

UGO DAL LAGO†, University of Bologna, Italy
ALEXIS GHYSELEN, University of Bologna, Italy

Model-checking is one of the most powerful techniques for verifying systems and programs, which since the pioneering results by Knapik et al., Ong, and Kobayashi, is known to be applicable to functional programs with higher-order types against properties expressed by formulas of monadic second-order logic. What happens when the program in question, in addition to higher-order functions, also exhibits algebraic effects such as probabilistic choice or global store? The results in the literature range from those, mostly positive, about nondeterministic effects, to those about probabilistic effects, in the presence of which even mere reachability becomes undecidable. This work takes a fresh and general look at the problem, first of all showing that there is an elegant and natural way of viewing higher-order programs producing algebraic effects as ordinary higher-order recursion schemes. We then move on to consider effect handlers, showing that in their presence the model checking problem is bound to be undecidable in the general case, while it stays decidable when handlers have a simple syntactic form, still sufficient to capture so-called *generic effects*. Along the way, we hint at how a general specification language could look like, this way justifying some of the results in the literature, and deriving new ones.

CCS Concepts: • **Theory of computation** → **Program verification**; **Verification by model checking**.

Additional Key Words and Phrases: higher-order recursion schemes, algebraic effects, model checking, effect handlers

ACM Reference Format:
Ugo Dal Lago and Alexis Ghyselen. 2024. On Model-Checking Higher-Order Effectful Programs. *Proc. ACM Program. Lang.* 8, POPL, Article 87 (January 2024), 29 pages. https://doi.org/10.1145/3632929

1 INTRODUCTION

Verifying the correctness of programs endowed with higher-order functions is a very challenging problem, which can be addressed with various methodologies, from type systems [Davies and Pfenning 2000; Freeman and Pfenning 1991; Hughes et al. 1996] to program logics [Brady 2013; Jung et al. 2018], from symbolic execution [King 1976; Tobin-Hochstadt and Van Horn 2012] to verified compilation [Leroy 2009]. An approach with some peculiarities is that of higher-order model checking (HOMC in the following), which consists in seeing the program at hand as a *structure*, then checking whether it renders a logical formula capturing the desired property true, namely whether it is a *model* of it. Saying it another way, HOMC can be seen as the application of the model checking paradigm [Clarke 1997; Clarke et al. 2018] to higher-order programs. One of the characteristics of this approach is that, contrary to most others, it is both sound *and complete*. As a consequence, the kind of languages to which the methodology can be applied are most often

*The authors are supported by the ERC CoG "Differential Program Semantics", GA 818616.
†Both authors contributed equally to this research.

Authors' addresses: Ugo Dal Lago, ugo.dallago@unibo.it, University of Bologna, Via Zamboni, 33, Bologna, BO, Italy, 40126; Alexis Ghyselen, alexis.ghyselen@unibo.it, University of Bologna, Via Zamboni, 33, Bologna, BO, Italy, 40126.

ACM 2475-1421/2024/1-ART87
https://doi.org/10.1145/3632929

not Turing-complete, the underlying verification problem being undecidable even for very simple logics.

The feasibility of higher-order model checking was scrutinized in the early 2000s, the objective being to extend classic results about model checking *recursion schemes* [Courcelle 1995] to higher-order generalizations of the latter. The results obtained were initially very interesting but partial [Knapik et al. 2001, 2002], only concerning certain restricted forms of higher-order recursion schemes. The quest came to an end in 2006 with Ong's groundbreaking result [Ong 2006] on the decidability of the model checking problem for trees generated by general higher-order recursion schemes against formulas of MSO (or, equivalently, of formulas the μ-calculus or alternating parity tree automata [Emerson and Jutla 1991; Grädel et al. 2003]). This result was followed by many other ones [Broadbent et al. 2010; Carayol and Serre 2012; Hague et al. 2008; Kobayashi 2009; Kobayashi and Ong 2009; Salvati and Walukiewicz 2014; Walukiewicz 2016], whose goal was that of understanding the deep computational nature of the problem, at the same time generalizing the decidability result and building concrete verification tools [Broadbent and Kobayashi 2013; Kobayashi 2011; Neatherway et al. 2012; Ramsay et al. 2014]. Readers are invited to refer to [Ong 2015] for an overview.

Among the various extensions to the HOMC problem considered in the literature, we should certainly mention extensions aimed at dealing with higher-order recursion schemes subject to more permissive type disciplines than that of simple types, namely the one to which Ong's classic result applies. As an example, higher-order recursion schemes with recursive types have been recently considered [Kobayashi and Igarashi 2013]. We should also mention some attempts at making the technique applicable to programs which are not pure, but which can produce, for example, nondeterministic choice, exception-handling and probabilistic effects. In the third case the decidability results scale back [Kobayashi et al. 2020], with undecidability showing up already at order two and for mere reachability properties. In the first case, instead results remain essentially unchanged [Kobayashi 2009; Tsukada and Kobayashi 2014], and in the second case, efficient verification tools can be obtained through optimized abstractions and transformations [Sato et al. 2013].

The motivation from which this work originates is precisely that of understanding the reasons for the aforementioned discrepancies, at the same time giving a general account of the HOMC problem in presence of effects. In doing so, we consider effects as captured by algebraic operations [Plotkin and Power 2003], the latter producing some pre-defined effects or interpreted by way of effect handlers [Hillerström et al. 2017; Kammar et al. 2013; Plotkin and Pretnar 2009]. This is not the first attempt at studying the verification of effectful higher-programs, as shown by fructuous approaches based on type and effect systems [Gordon 2020; Sekiyama and Unno 2023; Song et al. 2022]. In our work, we try to stay as general as possible, and thus we consider well-established ways of capturing effects in higher-order λ-calculi. On the side of specifications, we analogously try not to consider ad-hoc formalisms, and look for conservative extensions of MSO in which the properties of interest can be captured in a unifying way. In particular, in the algebraic approach to effects, trees are considered up to an equational theory describing how the different algebraic operations should behave. This means that it is particularly important to be able to define specifications that take into account this equational theory. This is captured in our framework by the notion of an *observation* [Johann et al. 2010; Matache and Staton 2019; Simpson and Voorneveld 2019].

The contributions of this paper are threefold:

- We first of all consider a finitary version of Simpson and Voorneveld's EPCF, a calculus with full recursion and algebraic effects, showing that the computation tree semantics of any EPCF program C is precisely the one of a λY-term C^* obtained by CPS-translating C. Since

the λY-calculus is well-known to be equiexpressive with higher-order recursion schemes [Nakamura et al. 2020], we obtain that the computation trees generated by EPCF can be automatically checked against MSO specifications. This is in Section 4 and Section 5.
- Then, we turn our attention to more general and more expressive specifications. Among the many proposals for a logic for algebraic effects, we consider a variation on the one proposed by Simpson and Voorneveld [Simpson and Voorneveld 2019], in which effectful computations can be tested through a notion of observation. We then prove that only certain notions of observation give rise to decidable model-checking problems, this way justifying some of the existing results in the literature, at the same time proving new ones. This is in Section 6.
- Finally, we consider the impact of effect handlers to the HOMC problem. We show that handlers are indeed harmful to decidability, at least when general, well-established notions of handlers are considered, including shallow and deep handlers. We conclude by considering a rather restricted class of handlers which are sufficiently expressive to capture generic effects [Plotkin and Power 2003], but for which model checking remains decidable. This is in Section 7 and Section 8.

All in all, the results above provide a rather clear picture about how far one can go in applying existing HOMC methodologies to effectful programs. The take-home messages are that in principle, such techniques can be reused, provided the underlying notion of observation does not give rise to too complex specifications, while handlers are potentially very dangerous, and should be used with great care. The technical core of the paper is in Section 3 to Section 8, while Section 2 serves as a gentle introduction to higher-order programming with effects and its verification. Related work is discussed in Section 9. Due to space limits, many details have been elided, but can be found in a longer version of this paper [Dal Lago and Ghyselen 2023].

2 HIGHER-ORDER EFFECTFUL PROGRAMS, AND HOW TO MODEL-CHECK THEM

While the λ-calculus is the reference paradigmatic model for pure functional programming, a standard way of raising *effects* from within functional programs consists in invoking algebraic operations [Plotkin and Power 2003], each of them corresponding to a particular way of producing an observable effect. Even when the underlying programming language does not offer algebraic operations natively, many impure constructs can be interpreted this way. Consider, as an example, the OCAML program in Figure 1a, call it P, which manipulates two ground global variables r and q through a recursive higher-order function f. As can be easily realized, whenever the conditional is executed the two references r and q contain *the same* boolean value. As a consequence, the Failure exception is never raised, and the program can be considered safe. Could we automatically verify the latter by way of higher-order model checking? Let us try to see if this is possible.

We can assume Loc to be a type of locations inhabited by r and q only, and that the program can invoke any effect-raising operations from the following typed signature:

$$\Sigma = \{\mathbf{Get} : \mathtt{Loc} \rightsquigarrow 2, \mathbf{Set} : \mathtt{Loc} \times \mathtt{Bool} \rightsquigarrow 1, \mathbf{Raise} : \mathtt{Unit} \rightsquigarrow 0\}.$$

where the numbers denote the *arity* of the operation. Some standard abbreviations allow us to form the term in Figure 1b, call it M_{P}, whose structure is very similar to the one of P. In doing so, we have adopted a syntax close to Simpson and Voorneveld's EPCF. Observe how OCAML's reference commands have become algebraic operations from Σ, and how **Get** has arity equal to two, accounting for the fact that the program can proceed depending on the value read from memory. By the way, M_{P} closely corresponds to the way one would write P in languages like EFF [Pretnar 2015].

But how could algebraic operations be helpful in the task of verifying the safety of P? Actually, the evaluation of programs which invoke algebraic operations naturally gives rise to a so-called

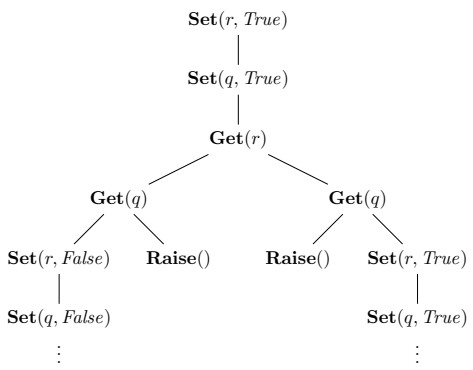
```
let r = ref true;;
let q = ref true;;
let rec f g =
  let y = !r in
  let z = !q in
  if (g y z) then failwith("Failure") else begin
    r := (not z);
    q := (not y);
    f g;
  end
in f (fun x y ↦ x <> y)
```

(a) An OCAML program P which never fails.

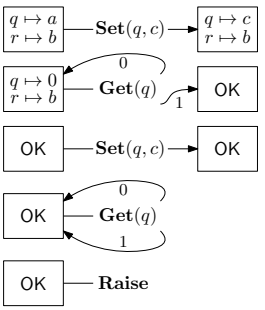
```
Set((r,true);λ_.
Set((q,true);λ_.
let F = return(fix f.λg.
  Get(r,λy.
  Get(q,λz.
  let x = (g y z) in
  case(x, Raise(),
    Set((r,not z);λ_.
    Set((q,not y);λ_.
    f g ))
  ))))
in F (λ(x,y). x ⊕ y)
))
```

(b) An EPCF term M_{P}.

Fig. 1. An OCAML program and its EPCF equivalent.



(a) Tree Generated by M_{P}.



(b) Automaton Capturing the safety of a program working with the boolean variables r and Q.

Fig. 2. Effect tree and automaton capturing safety.

effect tree. The effect tree produced in output by M_{P} looks like the one in Figure 2a. Through it, one can verify that **Raise** is never executed by exhaustively considering all branches of the tree, and verifying that those which are somehow coherent with the store operations do not end in a leaf labelled with **Raise**. Here, coherent branches are those that, e.g., when encountering $\mathbf{Get}(q)$ proceed left (respectively, right) depending on the last $\mathbf{Set}(q, a)$ operation performed. This reasoning can indeed be encoded by an alternating parity tree automata, and thus by a MSO or μ-calculus formula [Emerson and Jutla 1991; Grädel et al. 2003]. Let us briefly describe how this automaton can be constructed (here, by the way, we only need a top-down deterministic automaton). Its set of states comprises all assignments of boolean values to the variables r and q, together with a special accepting state OK. Intuitively, the latter captures incoherent states, and any action is accepted from there. Some example transitions are in Figure 2b. The first two capture the expected behaviour of **Set** and **Get** on stores, while the last three are there to model the fact that every action, including **Raise** is allowed in the state OK. Of course, **Raise** is *not* available in ordinary stores instead, since the property of interest is precisely the absence of uncaught exceptions.

All in all, then, we indeed have a way to turn impure functional programs into terms of a calculus called EPCF which generate trees that are, at least superficially, amenable to model checking. There is still a missing link though: it could well be that EPCF is simply too expressive, and that the model-checking problem is undecidable. Fortunately, however, we can turn EPCF programs into λY-terms, for which MSO model checking is indeed decidable [Nakamura et al. 2020]. This is precisely what we prove in Section 5 below.

Summing up, HOMC could indeed be helpful when performed on effect trees, because the program we are interested in verifying, and arguably any term working with global references over finite domains can be turned into a term of the λY-calculus, while the property becomes an MSO formula. But how about other effects? Can we turn the construction above into something more general and systematic? While specific kinds of effects are considered in the literature [Kobayashi 2009; Kobayashi et al. 2020; Ong 2015], no *general* result is known. In fact, each effect comes equipped with its intended notion of observation [Dal Lago et al. 2017; Johann et al. 2010; Matache and Staton 2019; Simpson and Voorneveld 2019]. Which ones of those are simple enough to guarantee that model checking useful properties stays decidable?

For the sake of convincing the reader that the questions above are not trivial, let us consider another example of an effectful higher-order program, borrowed from [Kobayashi et al. 2020]:

```
let F = return(fix f. λ_.
  Flip(λ b.
    case(b, return(),
      let x = f () in let y = f() in f()
))) in F ()
```

The algebraic operation $\mathbf{Flip}(\lambda b.C)$ should be understood as flipping a random coin, storing the result in b, and continuing as C. Intuitively, the program described above defines a procedure f that starts by flipping a coin. If the coin returns head, the procedure halts. Otherwise, the procedure is executed three times. This program generates an effect tree T_0 where for every n, the tree T_n can be defined as:

Indeed, the sequential composition in the third argument of the `case` operator gives raise to a stack of continuations, meaning that a leaf `return()` is replaced by the tree computed by the continuation. Here, the tree T_n should be understood as a call to the recursive function f with a continuation corresponding to n calls to this function. This behavior is similar to the one of a random walk, but the program described above does not use any infinite type, which is essential for Higher-Order Model Checking. In fact, this tree can be computed by a term of the λY-calculus, using a CPS translation, see Example 2.10 from [Kobayashi et al. 2020]:

$$(Y(\lambda F, k, x.\mathbf{Flip}\ (k\ x)\ (F\ (F\ (F\ k))\ x)))\ (\lambda x.x)\ \texttt{return}$$

Here, the use of higher-order functions is essential, since the function F takes a continuation as its first input, and this continuation determines the number of calls to the procedure that still need to be executed. For such a program, it is natural to wonder whether `return` is called or not (if the program terminates), but given the presence of (probabilistic) nondeterminism, there are various ways in which this can be spelled out. Do we mean that the program must (or may) reaches `return`? Or do we rather mean that the program reach `return` with probability 1? The latter question seems the most appropriate, given that probabilistic choice is captured by the subdistribution monad $\mathsf{D}(\cdot)$ and that $\mathsf{D}(\texttt{return})$ is just a real number between 0 and 1, i.e., the probability of not diverging.

$$\frac{}{\Gamma, x : T \vdash x : T} \qquad \frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x.M : T \to U} \qquad \frac{\Gamma \vdash M : T \to U \qquad \Gamma \vdash N : T}{\Gamma \vdash M\, N : U}$$

$$\frac{\Gamma \vdash M : T \to T}{\Gamma \vdash YM : T} \qquad \frac{}{\Gamma \vdash f : o^{\mathsf{ar}(f)} \to o}$$

Fig. 3. Typing Rules for λY-terms

If this is the case, however, recent results by Kobayashi, Dal Lago, and Grellois [Kobayashi et al. 2020] show that HOMC is *not* decidable in general, so the construction of an MSO formula (or an automaton) like the one above is simply not possible. May or must termination, instead, can easily be captured: may termination consists in exploring the tree and finding *at least* one `return`, while must termination is satisfied if the tree has *no* infinite branch, and all leaves are `return` leaves, which is a property that can be encoded as a parity condition of an alternating parity automaton.

To sum up, we can see effectful higher-order programs as producing effect trees computed by terms of the λY-calculus (or equivalently, higher-order recursion schemes) for which we know that MSO model-checking *is* decidable. This works particularly well for some effects, for example global store with finite domains, since the properties of interest can be expressed as an MSO formula. However, for some more involved effects, such as probabilistic choice, the underlying notion of observation is not expressible in MSO. The aim of this paper is to take a general look at this problem, and understand the deep reason behind this discrepancy.

3 PRELIMINARIES ABOUT HIGHER-ORDER MODEL CHECKING

3.1 Infinite Trees Generated by λY-Terms

In Higher-Order Model Checking, models are traditionally taken to be infinite trees produced by so-called Higher-Order Recursion Schemes (HORS in the following). In this work, we rather consider Böhm trees generated by ground type terms in the λY-calculus with first-order constants, which is well known to express the same class of trees as HORSs [Nakamura et al. 2020; Salvati and Walukiewicz 2012]. This section is devoted to presenting some preliminaries about the λY-calculus and the model checking problem for it.

Formally, the λY-calculus can be seen as the simply-typed λ-calculus extended with full recursion and with first-order function symbols:

$$\begin{array}{ll} \text{(Types)} & T, U ::= o \mid T \to U \\ \text{(Terms)} & M, N ::= x \mid \lambda x.M \mid M\, N \mid YM \mid f \in \Sigma \end{array}$$

The *signature* Σ is a set of first-order constants, such that each $f \in \Sigma$ comes equipped with an arity $\mathsf{ar}(f) \geq 0$ capturing the fact that the type of f is $\underbrace{o \to \cdots \to o}_{\mathsf{ar}(f) \text{ times}} \to o$, often abbreviated as $o^{\mathsf{ar}(f)} \to o$. When this does not cause ambiguity, we usually write $f : \mathsf{ar}(f)$ to specify the arity of f in a given signature . In this work, the type o should be understood both as the type of trees and as the answer type for the CPS translations we introduce below. Typing rules are standard, and can be found in Figure 3.

We see the λY-calculus as a tool to generate infinite trees. In order to precisely define the tree generated by a typable term, one has to define a form of dynamic semantics, which we here take as *weak head reduction*: we never reduce the argument of any application, and we never evaluate terms in the scope of λ-abstractions. Weak head reduction is traditionally the notion of reduction the λY-calculus is endowed with, and it makes the generation of infinite trees more natural, similarly

$$\frac{}{(\lambda x.M)N \to M[N/x]} \qquad \frac{}{YM \to M(YM)} \qquad \frac{M \to N}{M\ L \to N\ L}$$

Fig. 4. Weak Head Reduction Rules

to the reduction found in HORS. Rules for weak head reduction are standard, and can be found in Figure 4. It is relatively easy to see that typed closed terms in *weak head normal form*, i.e. typed terms with no free variables that cannot be further reduced, are precisely the (typable) terms generated by the following grammar:

$$J, K ::= \lambda x.M \mid f\ M_1\ M_2\ \cdots\ M_n$$

Indeed, a λ-abstraction is in normal form, constants are in normal form, and $M\ N$ is in normal form if and only if M is in normal form and M is not a λ-abstraction. As a consequence, we can easily realize that terms of ground type in weak head normal form have the shape $f\ M_1\ M_2\ \cdots\ M_{\mathsf{ar}(f)}$, where each M_i is itself a term of ground type. This naturally suggests a potentially infinite process turning any such term into a tree with root f and $\mathsf{ar}(f)$ subtrees obtained by evaluating $M_1\ \cdots\ M_{\mathsf{ar}(f)}$, respectively. This can be made formal as follows:

Definition 1 (Infinite Trees). *The set of (potentially infinite) trees generated by a signature Σ, denoted $\mathbf{Tree}_\Sigma$, is coinductively defined by the grammar:*

$$t ::= f(t_1, \cdots, t_{\mathsf{ar}(f)}) \mid \bot$$

where $f : \mathsf{ar}(f) \in \Sigma$.

The constant $\bot$ represents a non-terminating non-productive computation that does not generate any function symbol. As an example, if Σ is the signature $\{g : 2, f : 1, a : 0\}$, one can form the non-regular infinite tree $g(a, g(f(a), g(f(f(a)), \cdots))$. We are now ready to define how any closed ground λY-term generates such an infinite tree:

Definition 2 (Böhm Trees of Closed Ground Terms). *Given a closed term M of type o, i.e. we have $\cdot \vdash M : o$, the* Böhm tree *of M, denoted $BT(M)$, is defined by way of the following essentially infinitary process. Starting from M, we apply $\to$* ad infinitum. *This can have two possible outcomes:*

- *M can be reduced infinitely, and in this case $BT(M)$ is simply $\bot$*
- *M can be reduced to a weak head normal form $J = f\ M_1\ M_2\ \cdots\ M_{ar(f)}$ such that for all $1 \le i \le n$, we have $\vdash M_i : o$. Then, $BT(M) = f(BT(M_1), BT(M_2), \cdots, BT(M_{\mathsf{ar}(f)}))$.*

The aforementioned process is infinitary in two different ways: the evaluation of M can diverge, and $BT(M)$ can be infinite. The process above is well-defined only for closed terms of ground type and is thus less general than the one generating Böhm Trees for arbitrary terms of the λY-calculus [Clairambault and Murawski 2013] (there, in particular, one has to deal with λ-abstractions).

Example 1. *As an example, with the signature $\Sigma = \{g : 2, f : 1, a : 0\}$, consider the term*

$$M \triangleq (Y\ (\lambda F.\lambda x.g\ x\ (F\ (f\ x)))).$$

We pose $M_{step} \triangleq (\lambda F.\lambda x.g\ x\ (F\ (f\ x)))$. By applying weak head reduction rules, we obtain

$$M\ a \to M_{step}\ M\ a \to (\lambda x.g\ x\ (M\ (f\ x)))\ a \to g\ a\ (M\ (f\ a))$$

Thus, $BT(M\ a) = g(a, BT(M\ (f\ a)))$, and similarly $BT(M\ (f\ a)) = g(f(a), BT(M\ (f\ (f\ a))))$, finally we obtain that $BT(M\ a) = g(a, g(f(a), g(f(f(a)), \cdots)))$ the infinite tree described above.

3.2 Expressing Properties As Alternating Parity Tree Automata

Now that we have defined the class of *models* for higher-order model checking, we can define the specification language. The gold standard in model-checking consists in properties expressed by formulas of monadic second order logic [Knapik et al. 2001] (MSO for short). However, several equiexpressive specification languages have been defined in the literature, notably μ-calculus [Walukiewicz 1993] and alternating parity tree automata (APT for short) [Emerson and Jutla 1991; Grädel et al. 2003]. The latter is commonly used in the HOMC literature [Kobayashi and Ong 2009; Ong 2006, 2015], and as the examples we define through the paper have a simple representation as automata, we also chose this specification language.

Given a set X of variables, we define the *positive Boolean formulas* over X by way of the following grammar:

$$\phi, \psi ::= \mathtt{tt} \mid \mathtt{ff} \mid x \mid \phi \wedge \psi \mid \phi \vee \psi$$

with $x \in X$. The set of all positive Boolean formulas over X is indicated as $B^+(X)$. A subset $Y \subseteq X$ satisfies a formula ϕ, denoted $Y \vDash \phi$ if and only if interpreting any $x \in Y$ by $\mathtt{tt}$ and any $x \notin Y$ by $\mathtt{ff}$ makes ϕ true. We can now give the formal definition of an APT automaton:

Definition 3 (Alternating Parity Tree Automaton). *An* alternating parity tree automaton *is a tuple* $\mathcal{A} = (\Sigma, Q, \delta, q_i, \Omega)$ *where*

- Σ *is a signature, defining tree constructors with their arity.*
- Q *is a* finite *set of states, with* $q_i \in Q$ *the initial state.*
- δ *is the transition function, such that for each* $f \in \Sigma$, *we have* $\delta(q, f) \in B^+(\{1, \ldots, \mathtt{ar}(f)\} \times Q)$. *A translation* $\delta(q, f)$ *is thus a positive formula on a state assignment to children nodes.*
- $\Omega : Q \to \{0, \ldots, M\}$ *is the priority function, with* $M \geq 0$ *an integer.*

To define the acceptance condition of an automaton, we need to introduce the notion of positions in a tree.

Definition 4. *Let* Σ *be a set of tree constructors with maximal arity* A. *The set of all positions of a tree* t, *denoted* $dom(t)$, *is a set of words on the alphabet* $\Gamma = \{1, \ldots, A\}$ *defined as:*

$$dom(\bot) = \varepsilon; \qquad dom(f(t_1, \ldots, t_k)) = \{\varepsilon\} \cup 1 \cdot dom(t_1) \cup \ldots \cup k \cdot dom(t_k);$$

where $\cdot$ *is the concatenation operator for words. For a tree* t *and a position* $\alpha \in dom(t)$ *we define* $t(\alpha) \in \Sigma \cup \{\bot\}$ *by:*

$$\bot(\varepsilon) = \bot; \qquad f(t_1, \ldots, t_k)(\varepsilon) = f; \qquad f(t_1, \ldots, t_k)(n \cdot \alpha) = t_n(\alpha).$$

Informally, then $t(\alpha)$ *is the label of the node at position* α *in* t.

A *run-tree* of an automaton $\mathcal{A}$ over a tree $t \in \mathbf{Tree}_\Sigma$ is a tree with constructors in $dom(t) \times Q$. This run-tree must satisfy the following two constraints:

- The root is (ε, q_i), representing the root of t in the state q_i.
- For any node (α, q) of the run-tree, there is a set $S \subseteq \{1, \ldots, \mathtt{ar}(t(\alpha))\}$ which satisfies the transition $\delta(q, t(\alpha))$. And, for each $(i, r) \in S$, one of the children of (α, q) in the run-tree is precisely $(\alpha \cdot i, r)$.

Finally, we say that an infinite branch $(\varepsilon, q_i) \cdots (\alpha_k, q_k) \cdots$ of the run tree satisfies the *parity condition* if the largest priority that occurs infinitely often in $\Omega(q_i) \cdots \Omega(q_k) \cdots$ is even. A run-tree is *accepting* if every infinite path in it satisfies the parity condition. And of course, a tree $t \in \mathbf{Tree}_\Sigma$ is accepted by an automaton $\mathcal{A}$ if there exists an accepting run-tree for t. It is well-known [Emerson and Jutla 1991; Grädel et al. 2003] that acceptance from an APT automaton is equivalent to satisfying a Monadic Second-Order (MSO) formula: for every automaton $\mathcal{A}$, there exists an MSO formula on tree Ψ such that t is accepted by $\mathcal{A}$ if and only if Ψ is true on t, and conversely.

Example 2. *We define an automaton* $\mathcal{A} = (\{a : 2, b : 1, c : 0\}, \{q_0, q_1\}, \delta, q_0, \Omega)$ *such that* $\mathcal{A}$ *accepts a tree* t *if and only if for every path in* t, c *occurs eventually after* b *occurs. We pose:*

- *For all* $q \in \{q_0, q_1\}$,*we have that:*

$$\delta(q, a) = (1, q) \wedge (2, q); \qquad \delta(q, b) = (1, q_1); \qquad \delta(q, c) = \mathtt{tt}.$$

  This means, intuitively, that a node labelled with a *propagates the state to both of its children, seeing the letter* b *changes the state to* q_1 *and a leaf* c *is always accepted.*
- *It holds that* $\Omega(q_0) = 0$ *and* $\Omega(q_1) = 1$

By definition, if q_1 *occurs infinitely often in a branch of the run tree, then the largest priority is* 1 *in this branch, and the tree does not satisfy the parity condition. Thus, for this automaton to accept a tree* t, *we must have either finite branches (terminating with* c*) or infinite branches in which* b *does not occur (in order to stay in the state* q_0*). This corresponds indeed to the property that for every path in* t, c *occurs eventually after* b *occurs.*

We can now define the model-checking problem and give the main result from [Ong 2006].

Definition 5 (MSO Model-Checking Problem for the λY-Calculus). *Given a closed* λY*-term* M *with* $\vdash M : o$ *and an APT* $\mathcal{A}$, *is* $BT(M)$ *accepted by* $\mathcal{A}$ *?*

Theorem 1 (Decidability [Ong 2006, 2015]). *The MSO model-checking problem for the* λY*-calculus is decidable.*

The complexity of this problem, itself studied in [Ong 2006], is exponential, depending on the *order* of the λY-term (or, equivalently, the order of the higher-order recursion scheme [Nakamura et al. 2020]). The aim of our work is on decidability, so we will not take orders into account, but we plan to study complexity issues in the future, given that a finer complexity analysis of related problems is already available [Nakamura et al. 2020; Tsukada and Kobayashi 2014].

Thanks to this theorem, showing that MSO model-checking is decidable for a class of infinite trees only requires showing that this class of trees can be computed by λY-terms, and this can be done via effective program transformation to the λY-calculus, as we will see in many occasions starting from the next section.

4 HIGHER-ORDER PROGRAMS WITH EFFECTS

In this section, we introduce a calculus with algebraic operations and fixpoints, called EPCF. This language can be seen as a fine-grained call-by-value variation on Plotkin's PCF endowed with effect-triggering operations, as described in [Simpson and Voorneveld 2019]. We consider a finitary version of this language which differs from Simpson and Voorneveld's one only in minor ways. Evaluating programs producing effects in CBV is natural, as it allows arguments to produce effects *before* being passed to functions.

EPCF is built around two syntactic categories, namely that of *values*, which denote data or functions, and the one of *computations*, which are instead programs which potentially produce effects when evaluated. Terms and types for EPCF are as follows:

$$\begin{array}{lll}
\text{(Values)} & V, W ::= v \mid \underline{n} \mid x \mid \lambda x.C \mid \mathtt{fix}\ x.V \\
\text{(Computations)} & C, B, A ::= V\ W \mid \mathtt{return}(V) \mid \mathtt{let}\ x = C\ \mathtt{in}\ B \mid \sigma(V; x.C) \mid \\
 & \qquad\qquad \mathtt{case}(V; C_1, \ldots, C_k) \\
\text{(Types)} & T, U ::= \mathsf{B} \mid \mathsf{k} \mid T \to U
\end{array}$$

We suppose we are given a set of *finite* ground types, ranged over by B. Examples include the unit type and the type of booleans. We similarly assume a set $\mathcal{V}$ of constant values each of a ground

$$\frac{v : \mathsf{B} \in \mathcal{V}}{\Gamma \vdash v : \mathsf{B}} \qquad \frac{0 < n \leq k}{\Gamma \vdash \underline{n} : \mathsf{k}} \qquad \frac{\Gamma, x : T \to U \vdash V : T \to U}{\Gamma \vdash \mathtt{fix}\ x.V : T \to U} \qquad \frac{\Gamma \vdash V : T}{\Gamma \vdash \mathtt{return}(V) : T}$$

$$\frac{\Gamma \vdash C : T \qquad \Gamma, x : T \vdash D : U}{\Gamma \vdash \mathtt{let}\ x = C\ \mathtt{in}\ D : U} \qquad \frac{(\sigma : \mathsf{B} \rightsquigarrow k) \in \Sigma \qquad \Gamma \vdash V : \mathsf{B} \qquad \Gamma, x : \mathsf{k} \vdash C : T}{\Gamma \vdash \sigma(V; x.C) : T}$$

$$\frac{\Gamma \vdash V : \mathsf{k} \qquad (\Gamma \vdash C_i : T)_{1 \leq i \leq k}}{\Gamma \vdash \mathtt{case}(V; C_1, \ldots, C_k) : T}$$

Fig. 5. Typing Rules for EPCF

$$\frac{}{(\lambda x.C)\ W \to C[W/x]} \qquad \frac{}{(\mathtt{fix}\ x.V)\ W \to (V[(\mathtt{fix}\ x.V)/x])\ W}$$

$$\frac{}{\mathtt{case}(\underline{n}, C_1, \cdots, C_k) \to C_n}$$

$$\frac{}{\mathtt{let}\ x = \mathtt{return}(V)\ \mathtt{in}\ C \to C[V/x]} \qquad \frac{C \to C'}{\mathtt{let}\ x = C\ \mathtt{in}\ B \to \mathtt{let}\ x = C'\ \mathtt{in}\ B}$$

$$\frac{}{\mathtt{let}\ x = \sigma(V; y.C)\ \mathtt{in}\ D \to \sigma(V; y.\mathtt{let}\ x = C\ \mathtt{in}\ D)}$$

Fig. 6. Semantics for EPCF

type, ranged over by v, e.g. a unique constructor $() \in \mathcal{V}$ with its associated type `Unit`. We also assume finite enumeration types k to be present, each of them inhabited by the values $\underline{1}, \underline{2}, \ldots, \underline{k}$. We introduce enumeration types, which support pattern matching, just to keep them distinct from finite base types.

Algebraic operations have the shape $\sigma(V; x.C)$ where V is said to be a *parameter*, and C represents the *continuation* of the computation. The variable x is bound in C and represents the choice made by the algebraic operation. We suppose given a signature set Σ of symbols for algebraic operations, where each $\sigma \in \Sigma$ is given an arity $\mathsf{B} \rightsquigarrow k$, meaning that σ is an algebraic operation of arity k depending on a parameter of type B. In an operation of arity k, the type of x in the continuation C is k. In particular, defining k different continuations $C_1, \ldots, C_k$ depending on the possible values of x can be done by way of the computation $\sigma(V; x.\mathtt{case}(x, C_1, \ldots, C_k))$. As common in call-by-value calculi, the fixpoint operator $\mathtt{fix}\ x.V$ is a value and, without any loss of generality, is only attributed function types. Finally, the shape of computations is restricted in that the only way to compose computations is to use the `let` constructor. For example, if we want to apply the result of the computation D to the function computed by C, we need to define a computation

$$\mathtt{let}\ x = D\ \mathtt{in}\ \mathtt{let}\ y = C\ \mathtt{in}\ y\ x \qquad \text{or} \qquad \mathtt{let}\ y = C\ \mathtt{in}\ \mathtt{let}\ x = D\ \mathtt{in}\ y\ x$$

As can be seen from this example, the order of evaluation is imposed by sequencing, a property which turns out to be crucial in presence of effects, as we will see when looking at the semantics.

As for the EPCF type system, which is pretty natural and standard, we give some of the rules in Figure 5. The typing rule for algebraic operations stipulates that an effect does not change the type of the whole computation, which is arbitrary. Intuitively this is because calling an algebraic operation induces an effect σ, depending on the parameter V, and the computation will continue as C after producing this effect.

We now introduce the dynamic semantics of EPCF which is based on the reduction rules from Figure 6. The cornerstone of this relation is the last rule, allowing algebraic operations to "percolate out" of non-trivial evaluation contexts. Intuitively, this rule tells us that if we have a computation of the shape $\mathtt{let}\ x = C\ \mathtt{in}\ D$, and C is a computation producing an effect σ, then this effect is

immediately visible from the outside, its continuation now incorporating the `let` expression at hand. If we go back to the aforementioned example describing the composition of computations, namely the pair of terms

$$\texttt{let}\ x = D\ \texttt{in}\ \texttt{let}\ y = C\ \texttt{in}\ y\ x; \qquad\qquad \texttt{let}\ y = C\ \texttt{in}\ \texttt{let}\ x = D\ \texttt{in}\ y\ x;$$

we can see that in the first case, the effects thrown in D occur first, followed by the ones in C, and that the converse holds for the other case.

Example 3. *Let us consider a computation which flips a fair coin, then returning the exclusive or of the obtained value and itself (which is thus always going to be false). All this makes use of an algebraic operation* **Flip** *having arity* $\texttt{unit} \rightsquigarrow 2$ *and of a binary function* $\oplus$ *computing the exclusive or, and which can anyway be easily written as a term (with a small abuse of notation allowing function application with multiple arguments):*

$$\texttt{let}\ x = \mathbf{Flip}(\cdot; y.y)\ \texttt{in}\ (x \oplus x)$$

In a call-by-value setting, this program should first flip a coin, thus always returning $\underline{0}$ *in the end. Indeed, the dynamic semantics allows the* **Flip** *operation to be visible from the outside:*

$$\texttt{let}\ x = \mathbf{Flip}(\cdot; y.y)\ \texttt{in}\ (x \oplus x) \to \mathbf{Flip}(\cdot; y.\texttt{let}\ x = y\ \texttt{in}\ x \oplus x)$$

There are two observations one needs to make now. The first is that the continuation $\texttt{let}\ x = y\ \texttt{in}\ x \oplus x$ *returns* $\underline{0}$ *for all values of* y*, as expected, but since it occurs as the argument of* **Flip** *it cannot be further reduced, and its evaluation must be captured somehow differently. The second one is that in presence of effects, confluence is lost. If, indeed, we pass* $\mathbf{Flip}(\cdot; y.y)$ *unevaluated to the right-hand side, in a call-by-name fashion, we land on a totally different term, namely one which flips* two *coins, possibly returning* $\underline{1}$*:*

$$\texttt{let}\ x = \mathbf{Flip}(\cdot; y.y)\ \texttt{in}\ (x \oplus x) \to_{cbn} (\mathbf{Flip}(\cdot; y.y) \oplus \mathbf{Flip}(\cdot; y.y)).$$

The example above implicitly tells us that, similarly to what happens in the λY-calculus, terms of EPCF generate infinite trees that we commonly call *effect trees*, but that the dynamic semantics is not by itself taking care of the unfolding: indeed, typed closed computations in normal form in EPCF are either of the shape $\texttt{return}(V)$, which is expected, or $\sigma(V; x.C)$, meaning that evaluation apparently stops as soon as an algebraic operation is encountered. The effect tree of a computation is what we are looking for and is defined as follows:

Definition 6 (Effect Trees for EPCF Computations). *For an algebraic effect signature* Σ *and a type* T*, let us define the signature* Σ_T *as follows:*

$$\Sigma_T = \{\sigma : k + 1 \mid ((\sigma : \mathsf{B} \rightsquigarrow k) \in \Sigma)\} \cup \{\texttt{return}(V) : 0 \mid \vdash V : T\} \cup \{v : 0 \mid v \in \mathcal{V}\}$$

Note that Σ_T *is infinite in the general case, but that it is guaranteed to be finite when* $\mathcal{V}$ *is finite and* T *is a ground (and thus finite) type. The effect tree of a typed computation* $\vdash C : T$*, denoted* $ET(C)$*, is a tree in* $\mathbf{Tree}_{\Sigma_T}$ *defined by the following essentially infinitary process. First of all, reduce* C *ad infinitum, with one of* three *possible outcomes:*

- *If* C *can be reduced ad infinitum without reaching a normal form, then* $ET(C) = \bot$
- *If* $C \to^* \texttt{return}(V)$ *then* $ET(C) = \texttt{return}(V)$
- *If* $C \to^* \sigma(v; x.C)$ *with* $\sigma : \mathsf{B} \rightsquigarrow k$ *then* $ET(C) = \sigma(v, ET(C[x := \underline{1}]), \cdots, ET(C[x := \underline{k}]))$

This definition of an effect tree differs a bit from the one in the literature by the use of a special branch for parameters. Formally, standard effect trees [Matache and Staton 2019; Simpson and Voorneveld 2019] would be defined on the signature

$$\Sigma'_T = \{\sigma_v : k \mid ((\sigma : \mathsf{B} \rightsquigarrow k) \in \Sigma) \wedge (\vdash v : \mathsf{B} \in \mathcal{V})\} \cup \{\texttt{return}(V) : 0 \mid\!\vdash V : T\},$$

with the expected definition when seeing an effect $\sigma(v; x.C)$. Intuitively, operations are indexed by their parameters, and this is equivalent to what happens in our definition since parameters are taken from some finite type B. In particular, any MSO formula on a standard tree with indexed effects can be translated to an equivalent formula on effect trees with branching for parameters, see Section 6, Proposition 3 for more details. Anyway, we notice that this definition is similar to the notion of Böhm trees we gave in Definition 2. And indeed, we show that we can relate the notion of an effect tree from EPCF and the notion of a Böhm tree from the λY-calculus. Formally, we need to use a CPS translation similar to the one described in [Matache 2018; Matache and Staton 2019].

Example 4. *As an example, we can compute a tree with similarities with the one of Example 1 in the λY-calculus. We take $\Sigma = \{\sigma_f : \mathtt{Unit} \rightsquigarrow 1, \sigma_g : \mathtt{Unit} \rightsquigarrow 2\}$, and the EPCF computation*

$$C \triangleq \mathtt{fix}\ F.(\lambda x.\sigma_g((), n.\mathtt{case}(n;, x\ ()), F(\lambda z.\sigma_f((), \_.x\ ()))))).$$

For the sake of clarity, let us ignore all the parameters for algebraic operations, which do not bring anything interesting here since they are always equal to $()$. Then, we have that

$$C\ (\lambda z.\mathtt{return}(())) \to \sigma_g((\lambda z.\mathtt{return}(()))\ (), C\ (\lambda z'.\sigma_f((\lambda z.\mathtt{return}(()))\ ())\ ())).$$

And if we proceed further along the computation, we see that the effect tree t of $C\ (\lambda z.\mathtt{return}(()))$ is

$$t \triangleq \sigma_g(\mathtt{return}(()), \sigma_g(\sigma_f(\mathtt{return}(())), \cdots).$$

Note that since we evaluate computation following the call-by-value order, we need to use thunk functions, something which is not needed in the λY-calculus. As expected, in the translation from EPCF to λY-calculus, we have to take all of this into account, and CPS turns out to be what we need.

5 FROM EPCF TO THE λY-CALCULUS

It is now time to turn the informal arguments we made at the end of the last section into something more formal. Indeed, translating any EPCF computation into a term of the λY-calculus is possible, as we are going to show.

The fact that EPCF ground values, arities, and parameters are all taken from finite types implies that for every algebraic effect signature Σ, the signature Σ_Y defined as $\{v : 0 \mid v \in \mathcal{V}\} \cup \{\sigma : k + 1 \mid ((\sigma : \mathsf{B} \rightsquigarrow k) \in \Sigma)\}$ is itself finite. This enables the definition of a transformation scheme from EPCF to the λY-calculus with constants in Σ_Y, which we first give on types:

Definition 7 (CPS Type Transform). *For the sake of conciseness, we use $\neg T$ to denote the type $T \to o$. We pose:*

$$\mathsf{B}^* = o; \qquad \mathsf{k}^* = \neg(o^k); \qquad (T \to U)^* = T^* \to \neg U^* \to o.$$

We extend this function to contexts, with $\emptyset^* = \emptyset$ and $(\Gamma, x : U)^* = \Gamma^*, x : U^*$.

Observe how base types and enumeration types are translated in two essentially different ways, and that arrows are translated following the usual double-negation scheme. The following Definition instead captures how our translation works on *terms*. Its structure is heavily influenced by the fact that we are mapping a CBV calculus to a CBN calculus.

Definition 8 (CPS Term Transformation). *We define V^* and C^* with the intuition that if $\Gamma \vdash V : T$ and $\Gamma \vdash C : U$ then $\Gamma^* \vdash V^* : T^*$ and $\Gamma \vdash C^* : \neg\neg U^*$:*

$$v^* = v \text{ when } v \in \mathcal{V} \qquad (\lambda x.C)^* = \lambda x.C^*$$
$$x^* = x \qquad (\mathtt{fix} x.V)^* = Y(\lambda x.V^*)$$
$$\underline{n}^* = \lambda x_1, \ldots, x_k.x_n \qquad \mathtt{case}(V; C_1, \ldots, C_k)^* = \lambda c.V^*\ (C_1\ c)\ \cdots\ (C_k\ c)$$
$$(V\ W)^* = \lambda c.V^*\ W^*\ c \qquad (\mathtt{let}\ x = C\ \mathtt{in}\ B)^* = \lambda c.C^*\ (\lambda x.B^*\ c)$$
$$\mathtt{return}(V)^* = \lambda c.c\ V^*$$
$$\sigma(V; x.C)^* = \lambda c.\sigma\ V^*\ (C^*[x := \underline{1}^*]\ c)\ \cdots\ (C^*[x := \underline{k}^*]\ c)$$

Remark that this translation can be done because of our choice of defining effect trees with a dedicated branch for parameters: it is not possible at transformation-time to know exactly the value of the parameter V of an algebraic operation, but with our definition of effect trees, we can postpone the identification of V^* by just passing it as an argument. This translation is well-typed, as shown by the following lemma, whose proof is straightforward:

Lemma 1. *If $\Gamma \vdash V : T$ then $\Gamma^* \vdash V^* : T^*$ and if $\Gamma \vdash C : U$ then $\Gamma \vdash C^* : \neg\neg U^*$.*

Then, we would like to show that this translation is a simulation. We have to be careful however since weak-head reduction turns out to be insufficient for our purposes and has to be extended with *administrative* reduction rules [de Groote 1994; Hillerström et al. 2017; Plotkin 1975]. Formally, we introduce a new reduction relation $\rightarrow_{adm}$ by way of the following rules:

$$\frac{}{(\lambda c.M_i)\ c \rightarrow_{adm} M_i} \qquad \frac{f \text{ constant} \qquad (M_i \rightarrow_{adm} N_i)_{1 \leq i \leq n}}{f\ M\ M_1\ \cdots\ M_k \rightarrow_{adm} f\ M\ N_1\ \cdots\ N_k}$$

Intuitively, administrative reduction consists in applying a β-rule, possibly in the argument of an application (which is not possible in head reduction). This small set of rules is exactly what we need to have the simulation, and it should be intuitive that applying those rules has no impact on the Böhm tree of the underlying λY-term. Remark that we could have defined the transformation of a σ using $(\lambda x.C^*)\ \underline{1}^*$ instead of directly $C^*[x := \underline{1}^*]$ if we wanted to avoid using substitutions in the encoding. But this would induce more administrative reductions.

Lemma 2. *If $C \rightarrow B$ then for all continuation c, there is a term M such that $C^*\ c \rightarrow^+ M$ and $B^*\ c \rightarrow_{adm}^{\leq 2} M$.*

Proof. The proof can be done by induction on the structure of a proof of $C \rightarrow B$. All cases are straightforward and do not need administrative reduction except for the rule

$$\mathtt{let}\ x = \sigma(V; y.C)\ \mathtt{in}\ D \rightarrow \sigma(V; y.\mathtt{let}\ x = C\ \mathtt{in}\ D),$$

in which we do need administrative reductions, as we need to reduce the arguments of the constant σ in the transformation $\sigma(V; y.\mathtt{let}\ x = C\ \mathtt{in}\ D)^*\ c$:

$$\lambda c.\sigma\ V^*\ ((\lambda c.C^*[y := \underline{1}^*]\ (\lambda x.D^*\ c))\ c)\ \cdots\ ((\lambda c.C^*[y := \underline{k}^*]\ (\lambda x.D^*\ c))\ c)\ c$$
$$\rightarrow_{adm}^{\leq 2} \sigma\ V^*\ (C^*[y := \underline{1}^*]\ (\lambda x.D^*\ c))\ \cdots\ (C^*[y := \underline{k}^*]\ (\lambda x.D^*\ c))$$

□

This means, in particular, that if a computation C never reaches a normal form, then $C^*\ c$ does not, either. Indeed, the only case when we use administrative reductions is the one about the reduction rule involving sequencing and algebraic operations: if this rule can be used in a computation, then the computation will have a normal form, beginning with this effect. Then, by comparing the normal form of C and $C^*\ c$, we obtain the following theorem.

Theorem 2. *For every* $\vdash C : T$*, and for every* $\vdash c : T^* \to o$*, it holds that* $BT(C^*\ c)$ *equals* $ET(C)[\mathtt{return}(V) \leftarrow BT(c\ V^*)]$.

Thus, $BT(C^*\ c)$ perfectly simulates the effects produced by C, except for return values: instead of a leaf we have a tree representing the effect applied to this value by the continuation. It thus makes sense to talk about the *MSO model-checking problem for EPCF* as the task of verifying given $\vdash C : T$ and $\vdash c : T^* \to o$ as above, and given an APT automaton $\mathcal{A}$, whether $BT(C^*\ c)$ is accepted by $\mathcal{A}$. Using standard results on Higher-Order Model Checking, we indeed have that:

Corollary 1. *The MSO model-checking problem for EPCF is decidable.*

In particular, if the continuation c returns a new constant K, then $BT(C^*\ c)$ is equal to $ET(C)$ with all return leaves replaced by K. Also, if T is a ground type, then c can be taken as the identity, and we have exactly decidability of MSO model-checking for $ET(C)$. In general, the set of all possible values in the form $\mathtt{return}(V)$ is infinite, as they could be, e.g., λ-abstractions, but Ttheorem 2 provides us with some freedom, as we can always pick the continuation we want to discriminate between those values.

6 SPECIFICATIONS THROUGH OBSERVATIONS

In the previous section, we proved that EPCF computations can be translated into equivalent (in the sense of the tree they generate) λY-terms. But how can we actually *exploit* this result for the sake of verifying interesting properties about the effect tree generated by EPCF programs? Of course, we can automatically verify whether a given MSO formula has (the effect tree generated by) a given EPCF term as its model. But, is it the end of the story?

As already, mentioned, there are many proposals in the literature [Matache and Staton 2019; Plotkin and Pretnar 2008; Simpson and Voorneveld 2019] about *logics* specifically designed for the sake of expressing properties of interest for effectful (possibly higher-order) programs. In this paper we will somehow follow the path recently hinted at by Simpson and Voorneveld [Simpson and Voorneveld 2019].

Effect trees as we have used them so far are not subject to any *equational theory*, the latter often being associated to effects and justified by their monadic interpretation [Moggi 1988]. As an example, the following equations hold in the state monad, itself a way to interpreting the first example from Section 2:

$$\mathbf{Set}(\ell, \mathtt{true}, \_.\mathbf{Get}(\ell, y.C)) = \mathbf{Set}(\ell, \mathtt{true}, \_.C[y := \underline{1}]) \quad \mathbf{Set}(\ell, a, \_.\mathbf{Set}(\ell, b, \_.C)) = \mathbf{Set}(\ell, b, \_.C)$$

Some other equations includes distributive laws and equalities for program starting with **Get** first, see Example 2.2 in [Bauer 2019] for the full set. As pointed out in [Simpson and Voorneveld 2019], we can capture those equations between effect trees agnostically, without the need of explicitly referring to monads, through the notion of an *observation*, namely a set of *properties* of effect trees:

Definition 9 (Observations [Simpson and Voorneveld 2019]). *For a given set of algebraic operations* Σ*, we define a set of* observations *as a set, denoted* $\mathcal{O}$*, such that each* $o \in \mathcal{O}$ *is a subset of the effect trees of type* $\mathtt{unit}$*, i.e* $o \in \mathbf{Tree}_{\Sigma_{\mathtt{Unit}}}$

On top of a set of observations $\mathcal{O}$, Simpson and Voorneveld define a logic endowed with a modal operator of the form $o\phi$, where o is an element of $\mathcal{O}$ and ϕ is a formula. An effect tree satisfies $o\phi$ if it is among the trees in o *and* all its leaves satisfy ϕ. This way, the logic can *observe* the tree at hand through the lenses of the observations in $\mathcal{O}$, implicitly accounting for equations: it is necessary to define $\mathcal{O}$ such that all its elements are invariant by the desired set of equations. The resulting logic is quite powerful, being infinitary in nature, and turns out to precisely capture observational equivalence [Dal Lago et al. 2017] for (a calculus closely related to) EPCF. Noticeably,

most notions of effects, including nondeterministic, probabilistic and state effects can be captured this way [Simpson and Voorneveld 2019]. Thanks to this notion of observation, we can work on effect trees without taking the quotient by the algebraic equalities, and use instead equivalence for the aforementioned modal logic with observations, a choice which seems a natural setting for higher-order model checking of effectful programs.

The logic at hand, however, turns out to be strictly more powerful than MSO or the μ-calculus. This happens for two distinct reasons: on the one hand, it is essentially infinitary in nature, this way accounting for the fact that the target calculus has *countable* base types. On the other hand, the underlying set of observations O can be arbitrarily complex, thus injecting a huge discriminating power into the logic, something that MSO cannot do natively.

Motivated by all that, we address the following question in the rest of this section: can we express useful properties of effect trees as observations *from within* MSO itself? In other words, is it the case that for a given notion of effect, all the elements of O are captured by MSO formulas? This way, we could at least be sure, given the results from Section 5, that model-checking is decidable *relative to* O. This is precisely what motivates the following definition of an extension of MSO (or μ-calculus) with observations.

Definition 10 (Observation Predicate). *Suppose given a signature Σ for effect trees, containing a set of algebraic operations, and possibly some other constructors (that could come, e.g., from parameter values, or special constants of the λY-calculus). Fix a set of observations O. Let $t \in \mathbf{Tree}_\Sigma$, and P a subset of constructors all of them of arity 0. We define $t[\in P]$ as the tree obtained from t by replacing the leaves in P with* `return(())` *and all the other leaves with $\bot$. We next define, for each $o \in O$ and each P, a formula o_P, called an* observation predicate, *such that for any effect tree t, we have*

$$t \vDash o_P \text{ iff } t[\in P] \in o$$

Note that we have indeed that $t[\in P]$ is an effect tree of type `Unit`*, thus it can be an element of o. Write MSO_O for the conservative extension of MSO (on infinite trees) obtained by enriching the class of formulas with observation predicates.*

This notion of observation predicate having access to a set P is, thanks to finiteness, equiexpressive to the formulas $o\phi$ defined in [Simpson and Voorneveld 2019], in which ϕ is supposed to be a formula that separates the correct leaves from the incorrect ones in an *infinite* set of leaves. Here, because of finiteness, we have a finite number of such separations, thus we can take a set P instead of a formula ϕ.

MSO_O can thus be seen as a natural candidate for a logic in which to write specifications about higher-order effectful programs. But is the model checking problem emerging out of it decidable? There are at least two possible answers to this question:

- Either the observation predicates from O are definable in MSO *itself*, in this case MSO_O is equiexpressive with MSO, and the model checking problem remains decidable.
- Or those observation predicates are *not* MSO-definable, and we cannot conclude, with the concrete risk of landing into an undecidable verification problem.

We will now see some cases of effects and observations, each of them corresponding to one of the two cases above. Let us start from an example of effect which is hard to capture.

Example 5 (Probabilistic Observations). *The first counter-example to MSO-definability of observation predicates comes from probabilistic effects. The set of observations for probabilistic choice, with a unique operation* **Flip** : `unit` $\rightsquigarrow 2$*, is given by $O = \{o_{>q} \mid q \in \mathbb{Q}, 0 \leq q < 1\}$. Essentially, an observation predicate $o_{>q}$ means that the probability that the program terminates is greater than q. This is* not *a MSO-definable property on effect trees, and this happens for very good reasons: higher-order model checking for probabilistic HORSs is in general undecidable [Kobayashi et al. 2020].* □

Before looking at some positive examples, we need to deal with some little discrepancies between our definition of an effect tree and the literature [Simpson and Voorneveld 2019]. Indeed, as explained in Section 4, effect trees as introduced in Definition 6 turn out to be slightly different from the standard ones, and this was essential to define the translation presented in Section 5. However, we would like to work directly with standard effect trees, as we did in Section 2, since they are simpler and more customary. In fact, we can show that our definition is equivalent to the usual one, at least as far as MSO model-checking is concerned: we can translate any MSO formula (or APT automaton) on standard effect trees to a formula (or an APT automaton) on effect trees with a special branch for parameters. Formally, we have:

Definition 11 (Adding a branch for parameters). *For any tree t defined with the signature*

$$\Sigma = \{\sigma_v : k \mid ((\sigma : \mathsf{B} \rightsquigarrow k) \in \Sigma) \wedge (\vdash v : \mathsf{B} \in \mathcal{V})\} \cup S$$

where S is a set of possible other constructors (e.g. coming from a chosen continuation in the λY-calculus, according to Corollary 1), we define the tree t_p on the signature

$$\Sigma_p = \{\sigma : k+1 \mid ((\sigma : \mathsf{B} \rightsquigarrow k) \in \Sigma)\} \cup \{v : 0 \mid v \in \mathcal{V}\} \cup S$$

as the tree t in which all nodes $\sigma_v(t^1, \cdots, t^k)$ are replaced by the tree $\sigma(v, t_p^1, \cdots, t_p^k)$. This translation turns standard effect trees to trees as per Definition 6.

We can now show that any MSO formula (or APT automaton) that is definable on $\mathbf{Tree}_\Sigma$ is also definable on $\mathbf{Tree}_{\Sigma_p}$:

Proposition 3 (Effect Trees and APT Automata). *For any APT automaton $\mathcal{A}$ that recognizes* standard *effect trees in $\mathbf{Tree}_\Sigma$, there exists an APT automaton $\mathcal{A}_p$ that recognizes the set $\{t_p \in \mathbf{Tree}_{\Sigma_p} \mid t \text{ is accepted by } \mathcal{A}\}$*

Proof. The proof consists in adding a special state q_v for each constant value $v \in \mathcal{V}$ and define the transition function as a disjunction of formulas depending on the value of the parameter. Formally, we define $\mathcal{A}_p = (\Sigma, Q_p, \delta_p, q_i, \Omega_p)$ such that:

- $Q_p \triangleq Q \sqcup \{q_v \mid v \in \mathcal{V}\}$
- $\Omega_p(q) = \Omega(q)$ for $q \in Q$ and $\Omega_p(q_v) = 0$.
- For the transition function we have:
  - $\delta_p(q, s) = \delta(q, s)$ if $s \in S$
  - For all $\sigma : \mathsf{B} \rightsquigarrow k$, we have $\delta_p(q, \sigma) = \bigvee_{v:\mathsf{B} \in \mathcal{V}} (\delta(q, \sigma_v)[+1] \wedge (1, q_v))$ where $\phi[+1]$ is the formula ϕ in which all pairs (n, r) are replaced by $(n+1, r)$.
  - For all $v \in \mathcal{V}$, $\delta_p(q_v, v) = \mathtt{tt}$ and $\delta_p(q_v, f) = \mathtt{ff}$ for all $f \neq v$.

Thus, intuitively, when we reach a node σ, only one of the $(1, q_v)$ will be satisfied, namely the one corresponding to the actual parameter v of σ. And then, the only satisfiable formula in this big disjunction would be $\delta(q, \sigma_v)[+1] \wedge (0, q_v)$, meaning that the transition is logically equivalent to $\delta(q, \sigma_v)[+1]$. Thus, we can see that the new transition $\delta_p(q, \sigma)$ is equivalent to the actual formula $\delta(q, \sigma_v)[+1]$ where v is the parameter of σ in the tree t_p, and from this it is straightforward to see that $\mathcal{A}_p$ recognizes the set $\{t_p \mid t \text{ is accepted by } \mathcal{A}\}$. □

Let us now turn to examples of observations which are indeed MSO-definable.

Example 6 (Exceptions). *Let us consider the singleton signature $\{\mathbf{Raise} : \mathtt{Unit} \rightsquigarrow 0\}$. By definition, an effect tree in this case is just a leaf, that is either* **Raise**, *$\bot$ or another leaf ℓ. The set of observations [Simpson and Voorneveld 2019] is then defined as*

$$O \triangleq \{\downarrow, \mathbf{E}\}$$

where the observation predicate $\downarrow_P$ means that the program terminates (the tree is not $\bot$) with a leaf in P, and $\mathbf{E}_P$ means that the tree is **Raise**, *and P is ignored. In this simple case, the observations just distinguish between leaves, which is something that can obviously be captured by MSO. By the way, this effect can easily be composed with other effects.* □

Example 7 (Nondeterminism). *Let us consider the singleton set of algebraic operations* $\{\mathbf{or} : \mathtt{Unit} \rightsquigarrow 2\}$, *which is enough to model binary non-deterministic choice. The notion of observation for nondeterminism is given by the two modalities $\Diamond$ and $\Box$, where $\Diamond_P$ is the set of effect trees with at least one finite branch ending a leaf in P, and $\Box_P$ is the set of effect trees with only finite branches, all of them having a leaf in P. As a remark, the usual equational theory associated with nondeterminism is the one of semi-lattices:*

$$C \mathbf{\ or\ } C = C \qquad C \mathbf{\ or\ } D = D \mathbf{\ or\ } C \qquad C \mathbf{\ or\ } (D \mathbf{\ or\ } E) = (C \mathbf{\ or\ } D) \mathbf{\ or\ } E$$

It should be clear that if two trees are observationally equivalent, i.e. they have the same truth value for all observational predicates, then they are in the same equivalence class modulo this algebraic theory. Moreover, the converse also holds. This is why we can use observations without the need to take the aforementioned equations into account. It can be shown that these observations are definable by APT automata. For the sake of the example, we explain informally the encoding of $\Box_P$. We take a unique state q_i. To impose finite branching everywhere, it is sufficient to pose $\Omega(q_i) = 1$, because this way no infinite branch satisfies the parity condition. Then, we can just explore the tree, with the transition $\delta(q_i, or) = (0, q_i) \wedge (1, q_i)$. And finally, we can separate leaves with $\delta(q_i, a) = \mathtt{tt}$ if $a \in P$ and $\delta(q_i, a) = \mathtt{ff}$ if $a \notin P$ for all leaves a. □

Example 8 (Finite State Monad). *Going back to the examples from Section 2, we start with the state monad, where the operations are* **Get** *and* **Set**, *with finite domains. Following [Simpson and Voorneveld 2019], the set of observations for this effect can be defined as follows:*

$$O \triangleq \{o_{p \mapsto q} \mid p, q \text{ are memory states}\}$$

with the intuitive semantics that $o_{p\mapsto q}$ contains all effect trees mapping the state p to the state q (this computation is deterministic because the initial state is fixed, so we always know which branch should be taken on a **Get**). *Again, we can verify that observations are as discriminating as equality for the algebraic theory of finite states. It is also easy to see that the observation predicates coming from this set are definable as APT automata. An observation predicate $(o_{p\mapsto q})_P$ can be represented as an automaton starting from a state q_p, that follows the only deterministic path induced by the configuration of the memory states, and accepts only when this leads to a leaf $x \in P$ in the state q_q, similarly to the automata defined in Section 2. Thus, as expected, we can deduce that many interesting properties can be decided on programs using a finite state monad, since observations are MSO-definable.* □

Example 9 (Input-Output). *To conclude this section, we present an example dealing with I/O effects. More specifically, we assume programs can interact with the environment by reading and writing boolean values. The underlying signature is thus* $\Sigma = \{\mathbf{Read} : \mathtt{Unit} \rightsquigarrow 2, \mathbf{Write} : \mathtt{Bool} \rightsquigarrow 1\}$. *This is actually the same signature as the one for the finite state monad when only* one *location is around. Since the equational theories of the two effects differ, the two effects give rise to distinct sets of observations. Following [Simpson and Voorneveld 2019], we first define an* I/O trace *as a word over the alphabet* $\{?0, ?1, !0, !1\}$, *representing an input/output sequence where $?b$ means that the boolean value b is taken in input, while $!b$ means that the program outputs b. The set of observations turns out to be*

$$O \triangleq \{\langle w\rangle\downarrow, \langle w\rangle\uparrow \mid \ w \text{ is an I/O trace}\}.$$

The observation $\langle w\rangle\downarrow_P$ is satisfied on t if w is a complete I/O trace of t resulting in termination with a leaf in P. Similarly, $\langle w\rangle\uparrow_P$ is satisfied on t if w is an initial I/O trace of t (and P is ignored). Again, it is

easy to see that any such observation predicate can be captured as an APT automaton, as w determines the branches that need to be followed. □

7 A CALCULUS OF EFFECTS AND HANDLERS

An algebraic operation in a language such as EPCF comes out *uninterpreted*: a computation produces an effect tree, in which all algebraic operations are tree constructors, and no meaning is given to operations in this tree. The notion of observation can indeed be seen as a way to at least identify trees which should be considered the same, implicitly capturing equations between them. Observations, however, attribute a *static* meaning to programs, and the programmer has no control over them. There is, however, yet another way of attributing meaning to effects in which the programmer has direct control on what happens. This can be seen as an abstraction of the notion of an exception handler, but also of, e.g., the HASKELL's monad construction. We are referring, of course, to the so-called *effect handlers* [Kammar et al. 2013].

In this section, we present a calculus obtained by endowing EPCF with effect handlers, called HEPCF, whose definition closely follows the literature on the subject [Hillerström et al. 2017; Kammar et al. 2013]. This is necessary to even understand what doing model checking actually means in presence of handlers, this way paving the way for positive and negative results about it, which are deferred to the next section.

The language HEPCF differs from EPCF both at the level of terms and at the level of types. The former are extended with the `handle` clause, in which a computation is evaluated in a protected environment, in such a way as to be able to capture (and handle) the effects it raises. The latter are enriched with annotations keeping track of which ones among the operations are visible. The syntax of HEPCF is defined as follows:

$$
\begin{array}{lr}
\text{(Types)} & T, U ::= \mathsf{B} \mid k \mid T \to_E U \\
\text{(Effects)} & E ::= \emptyset \mid \{\sigma : \mathsf{B} \rightsquigarrow k\} \cup E \\
\text{(Values)} & V, W ::= v \mid \underline{n} \mid x \mid \lambda x.C \mid \mathtt{fix}\, f.V \\
\text{(Computations)} & C, D ::= V\ W \mid \sigma(V; x.C) \mid \mathtt{return}(V) \mid \mathtt{let}\ x = C\ \mathtt{in}\ D \\
 & \qquad \mid \mathtt{with}\ H\ \mathtt{handle}\ C \mid \mathtt{case}(V; C_1, \ldots, C_k) \\
\text{(Handlers)} & H ::= \{\mathtt{return}(x) \mapsto C_r\} \cup \{\sigma_i(x; r) \mapsto C_i \mid 1 \leq i \leq n\}
\end{array}
$$

The arrow type $T \to_E U$ refers to the set E of available algebraic operations the function at hand can possibly perform once applied to an argument of type T. An handler consists of a return clause $\mathtt{return}(x) \mapsto C_r$ and of one clause $\sigma_i(x; r) \mapsto C_i$ for each handled operation σ_i in some effect set E. The intuition behind such an handler is that it takes a computation using the effects in $\{\sigma_i \mid 1 \leq i \leq n\}$, and interprets each call to the algebraic operation σ_i as the computation C_i in which the continuation is passed as a variable r. The return computation C_r is called when the initial computation returns a value.

Selected rules of the type system are in Figure 7. A judgment attributing a type to a computation C now has the shape $\Gamma \vdash_E C : T$, meaning that C has the type T in the effect context E. As the reader can see in the rule dealing with calls to algebraic operations, a computation of type $\Gamma \vdash_E C : T$ can only give rise to the operations in E. As expected, a handler takes a computation C of some type U that uses the effects in F and transforms this computation into a computation of type T using the effects in E. This operation is very similar in principles to an application, that is why we describe the type of an handler with an arrow type. In details, the type $U_F \Rightarrow_E T$ means that all the computations defined in the typed handler should use only the effect in E, and the handled operations are exactly those in F. In such a context, a continuation for an algebraic operation is

$$\frac{\Gamma, x : T \vdash_E C : U}{\Gamma \vdash \lambda x.C : T \to_E U} \qquad \frac{\Gamma, f : T \to_E U \vdash V : T \to_E U}{\Gamma \vdash \mathtt{fix}\ f.V : T \to_E U}$$

$$\frac{\Gamma \vdash V : T \to_E U \qquad \Gamma \vdash W : T}{\Gamma \vdash_E V\ W : U} \qquad \frac{(\sigma : \mathsf{B} \rightsquigarrow k) \in E \qquad \Gamma \vdash V : \mathsf{B} \qquad \Gamma, x : k \vdash_E C : T}{\Gamma \vdash_E \sigma(V; x.C) : T}$$

$$\frac{\Gamma \vdash V : T}{\Gamma \vdash_E \mathtt{return}(V) : T} \qquad \frac{\Gamma \vdash_E C : U \qquad \Gamma, x : U \vdash_E D : T}{\Gamma \vdash_E \mathtt{let}\ x = C\ \mathtt{in}\ D : T}$$

$$\frac{\Gamma \vdash V : k \qquad (\Gamma \vdash_E C_i : T)_{1 \le i \le k}}{\Gamma \vdash_E \mathtt{case}(V; C_1, \ldots, C_k) : T} \qquad \frac{\Gamma \vdash H : U_F \Rightarrow_E T \qquad \Gamma \vdash_F C : U}{\Gamma \vdash_E \mathtt{with}\ H\ \mathtt{handle}\ C : T}$$

$$\frac{F = \{\sigma_i : \mathsf{B}_i \rightsquigarrow k_i \mid 1 \le i \le n\} \qquad \Gamma, x : U \vdash_E C_r : T \qquad \Gamma, x : \mathsf{B}_i, r : k \to_E T \vdash_E C_i : T}{\Gamma \vdash \{\mathtt{return}(x) \mapsto C_r\} \cup \{\sigma_i(x; r) \mapsto C_i \mid 1 \le i \le n\} : U_F \Rightarrow_E T}$$

Fig. 7. Static Semantics of HEPCF

$$\frac{C \to D}{\mathtt{with}\ H\ \mathtt{handle}\ C \to \mathtt{with}\ H\ \mathtt{handle}\ D}$$

$$\frac{H = \{\mathtt{return}(x) \mapsto C_r\} \cup \{\sigma_i(x; r) \mapsto C_i \mid 1 \le i \le n\}}{\mathtt{with}\ H\ \mathtt{handle}\ \mathtt{return}(V) \to C_r[x := V]}$$

$$\frac{H = \{\mathtt{return}(x) \mapsto C_r\} \cup \{\sigma_i(x; r) \mapsto C_i \mid 1 \le i \le n\}}{\mathtt{with}\ H\ \mathtt{handle}\ \sigma_i(V; x.C) \to C_i[x := V, r := (\lambda x.\mathtt{with}\ H\ \mathtt{handle}\ C)]}$$

Fig. 8. Dynamic Semantics for HEPCF

given as the variable r of type $k \to_E T$, meaning that this continuation is, as expected, of type T with effects in E, and there are k different continuations of this type, corresponding to the k possible branches of this algebraic operation.

Finally, the dynamic semantics of HEPCF is described in Figure 8 and Figure 6. As expected, handling a returned value consists in calling the return clause, and for the case of an algebraic operation, it is important to note that the handler acts also on the continuation. As for the definition of an effect tree, we can see that as before, a closed typed computation $\vdash_E C : T$ in normal form is either an algebraic operation in E or $\mathtt{return}(V)$, thus we can define effect trees similarly to Definition 6, with the signature E instead of Σ. Note that we can see the typing $\vdash H : T_F \Rightarrow_E U$ as defining a tree operation from effect trees on F of type T to effect trees on E of type U. In practice, this transformation takes a tree defined by $\vdash_F C : T$ and gives a tree defined by $\vdash_E \mathtt{with}\ H\ \mathtt{handle}\ C : U$. We will see in the next section that, unfortunately, the tree transformations that can be expressed by handlers form a very large class, definitely too broad for our purposes.

8 MODEL CHECKING HANDLED EFFECTS: POSITIVE AND NEGATIVE RESULTS

In the previous section, we introduced effect handlers, a very powerful and elegant linguistic construction thanks to which the interpretation of algebraic operations can somehow be delegated to the programmer. In this section, we show that handlers are simply *too* expressive, meaning that MSO model-checking of effect trees produced by terms of HEPCF is *not* decidable. We will also show that when handlers have a restricted but non-trivial form, model checking becomes decidable.

8.1 Undecidability Through the Halting Problem

The proof of undecidability is structured around the encoding of Plotkin's PCF into HEPCF. The encoding is somehow challenging, since PCF is taken in its full generality (thus including a type of natural numbers), while HEPCF only has *finite* base types. Since the halting problem for PCF is well-known to be undecidable, and termination can be written down as an MSO formula, undecidability of the HOMC problem for HEPCF easily follows. Formally, we introduce standard PCF as the following language:

Definition 12 (Plotkin's PCF). *The PCF language (in fine-grained call-by-value) is defined by the following grammar:*

$$\begin{array}{lll} \textit{(Types)} & & T ::= \mathsf{Nat} \mid T \to U \\ \textit{(Values)} & & V, W ::= 0 \mid \mathtt{succ}(V) \mid x \mid \lambda x.C \mid \mathtt{fix}\ f.V \\ \textit{(Computations)} & & C, D ::= V\ W \mid \mathtt{return}(V) \mid \mathtt{let}\ x = C\ \mathtt{in}\ D \mid \mathtt{case}(V; 0 \mapsto C; \mathtt{succ}(x) \mapsto D) \end{array}$$

This is a well-known language for which the halting problem is not decidable: encoding all partial recursive functions is an easy exercise. We omit the definition of the static and dynamic semantics, which are anyway natural and very standard.

In order to prove the undecidability of MSO-model checking for HEPCF we give a translation from PCF to HEPCF. Again, note that the crucial difference between the two languages is that PCF has access to an infinite base type Nat for natural numbers, with a pattern matching constructor, whereas in HEPCF there are handlers and algebraic effects, but all base types must be finite. But how could we encode the infinite type of natural numbers using finite types, effects and handlers? In order to do so, we introduce the effect $E = \{\sigma : \mathtt{Unit} \rightsquigarrow 1\}$. For the sake of simplicity, we ignore the parameter for this operation, of type Unit. Intuitively, we represent a natural number n by a computation in which this operation σ is called exactly n times, and then the computation halts by returning the only inhabitant of the type Unit. But, because a natural number n is supposed to be a value, we will use a thunk function f_n along the encoding, and each time we need to inspect the value of a natural number n, we can call $f_n()$ to produce the effect tree corresponding to the tree representation of this number n.

Definition 13 (Translation from PCF to HEPCF). *For any type T of HEPCF we define the* zero *of this type, denoted Z_T as follows:*

$$\mathsf{Z}_{\mathtt{Unit}} = (); \qquad \mathsf{Z}_k = \underline{1}; \qquad \mathsf{Z}_{T \to_E U} = \lambda x.\mathtt{return}(\mathsf{Z}_U).$$

Intuitively, the zero of a type is just a particular closed value of this type chosen arbitrarily, in which no effects are used. For the sake of clarity, we introduce a notation for several λ-abstractions and several applications in a row:

$$\lambda f, g.C \triangleq \lambda f.\mathtt{return}(\lambda g.C) \qquad V\ W\ U \triangleq \mathtt{let}\ x = V\ W\ \mathtt{in}\ (x\ U)$$

for which we obtain, as expected, that $(\lambda f, g.C)W\ U \to^+ C[f := W][g := U]$. The translation $[\![\cdot]\!]$ from PCF to finitary HEPCF is then given by

$$[\![\mathsf{Nat}]\!] = \mathtt{Unit} \to_E \mathtt{Unit} \qquad [\![T \to U]\!] = [\![T]\!] \to_E [\![U]\!]$$

$$\begin{array}{rlrl} [\![0]\!] & \triangleq \mathsf{Z}_{[\![\mathsf{Nat}]\!]} & [\![\mathtt{succ}(V)]\!] & \triangleq \lambda x.\sigma(y.[\![V]\!]()) \\ [\![x]\!] & \triangleq x & [\![\lambda x.C]\!] & \triangleq \lambda x.[\![C]\!] \\ [\![\mathtt{fix} f.V]\!] & \triangleq \mathtt{fix} f.[\![V]\!] & [\![V\ W]\!] & \triangleq [\![V]\!]\,[\![W]\!] \\ [\![\mathtt{return}(V)]\!] & \triangleq \mathtt{return}([\![V]\!]) & [\![\mathtt{let}\ x = C\ \mathtt{in}\ D]\!] & \triangleq (\mathtt{let}\ x = [\![C]\!]\ \mathtt{in}\ [\![D]\!]) \end{array}$$

$$[\![\mathtt{case}(V; 0 \mapsto C; \mathtt{succ}(n) \mapsto D)]\!] \triangleq \mathtt{let}\ a = (\mathtt{with}\ H(C,D)\ \mathtt{handle}\ [\![V]\!]\ ())\ \mathtt{in}\ (a\ (\lambda f, g.g\ ()))$$

where $H(C,D)$ is defined, when C and D have type T, by:

$$\begin{aligned}\{\ \mathtt{return}(V) \mapsto\ & \mathtt{return}(\lambda p.p\ \mathsf{Z}_{\mathtt{Unit}\to_E \mathtt{Unit}}\ (\lambda x.[\![C]\!])),\\ \sigma(r) \mapsto\ & \mathtt{let}\ n = \mathtt{return}(\lambda x.\mathtt{let}\ y = r\ \underline{1}\ (\lambda f, g.\mathtt{let}\ z = f\ ()\ \mathtt{in}\ \mathsf{Z}_{[\![T]\!]})\ \mathtt{in}\\ & \qquad \mathtt{return}(\mathsf{Z}_{\mathtt{Unit}}))\ \mathtt{in}\\ & \mathtt{return}(\lambda p.p\ (\lambda x.\sigma(z.n\ ()))\ (\lambda x.[\![D]\!]))\ \}\end{aligned}$$

Let us now give some hints about the translation. As expected, zero is mapped to the function that returns () when called, while the successor becomes a call to the effect σ. In order to do the pattern matching for a natural number V given as a thunk function $[\![V]\!]$, we produce the effect tree representing n, by calling $[\![V]\!]()$, and handling it. This handler produces a pair of thunked computations. The first computation is just a copy of the initial computation $[\![V]\!]()$. This is rather obvious for the return clause, and for the case of σ, the value n corresponds intuitively to just taking the first computation in r, thus a copy of the predecessor, and then the first computation is σ applied to this n, which gives the successor of the predecessor, i.e. a copy of the current number. The second computation simulates the initial pattern matching. For the case of a zero, the second computation is just C, and for the case of a successor (an operation σ), the second computation takes the copy of the predecessor given by n, and then returns the computation D that uses this predecessor.

The proof that this translation is correct is inspired by the proof of the encoding of shallow handlers using deep handlers, described in [Hillerström and Lindley 2018]. Intuitively, this is because a shallow handler is a handler that only handles the root of an effect tree, which is exactly what a pattern matching is supposed to do. First, we can show that this translation is well-typed:

Lemma 3. *If $\Gamma \vdash V : T$ then $[\![\Gamma]\!] \vdash [\![V]\!] : [\![T]\!]$ and if $\Gamma \vdash C : T$ then $[\![\Gamma]\!] \vdash_E [\![C]\!] : [\![T]\!]$*

Proof. The proof is direct except for the `case` construct, where we need to prove this intermediate result:

If $\Gamma \vdash C : T, \Gamma, n : \mathtt{Nat} \vdash D : T$, $[\![\Gamma]\!] \vdash [\![C]\!] : [\![T]\!]$ and $[\![\Gamma]\!], n : [\![\mathtt{Nat}]\!] \vdash [\![D]\!] : [\![T]\!]$, then

$$[\![\Gamma]\!] \vdash H : \mathtt{Unit}_E \Rightarrow_E ((\mathtt{Unit} \to \mathtt{Unit}) \to (\mathtt{Unit} \to [\![T]\!]) \to [\![T]\!]) \to [\![T]\!]$$

With this type in mind, the proof that $H(C,D)$ inhabits this type is straightforward, and we can conclude that the translation for `case` is well-typed. □

We want to prove that the translation is a correct simulation. In order to do this, we need to introduce formally an approximation of a term up to some administrative context, as in [Hillerström and Lindley 2018]: in this translation we encode natural numbers as functions, and we sometimes have a term that is overly complex because all the computation is hidden behind a λ, but this term will behave exactly like another simple term corresponding to the encoding of a natural number. This is exactly what happens here; the variable n in the handler is essentially a copy of the predecessor, but syntactically n is described as a complex computation.

Definition 14 (Administrative Contexts (See [Hillerström and Lindley 2018],Definition 6)). *Evaluation contexts are defined by the following grammar:*

$$\mathcal{E} ::= [\,] \mid \mathtt{let}\ x = \mathcal{E}\ \mathtt{in}\ C \mid \mathtt{with}\ H\ \mathtt{handle}\ \mathcal{E}$$

They are precisely those contexts in which computation can take place. An evaluation context $\mathcal{E}$ for a computation is called administrative*, denoted adm($\mathcal{E}$) if and only if:*

- *For all values V, $\mathcal{E}[\mathtt{return}(V)] \to^* \mathtt{return}(V)$*

- *For all operations* σ, $\mathcal{E}[\sigma(V; x.C)] \to^* \sigma(V; x.D)$ *with* $D \to^* \mathcal{E}[C]$.

Remark that the composition of two administrative contexts is also an administrative context.

Definition 15 (Approximation up to Administrative Reductions (See [Hillerström and Lindley 2018],Definition 7)). *We define* $\geq$ *for terms as the closure under* any *context of the following rules:*

$$\frac{}{C \geq C} \qquad \frac{adm(\mathcal{E}) \qquad C \geq D}{\mathcal{E}[C] \geq D} \qquad \frac{C \to E \qquad E \geq D}{C \geq D}$$

Intuitively, we have $C \geq D$ *when* C *could be reduced to* D *using strong rules of reduction (reducing under any context) without changing normal forms (of the form* `return` *or* σ*).*

We can then prove the following lemma about the just introduced relation, showing that the properties of administrative contexts are valid for approximations.

Lemma 4. *If* $D \geq C$ *then:*

- *If* C *is* $\mathtt{return}(V)$*, then* $D \to^* \mathtt{return}(W)$ *with* $W \geq V$
- *If* C *is* $\sigma(V; y.E)$*, then* $D \to^* \sigma(W; y.F)$ *with* $F \geq E$

Proof. By induction on $D \geq C$, it can be found in [Dal Lago and Ghyselen 2023]. □

We can then show that this approximation is a simulation.

Lemma 5. *If* $\Gamma \vdash C : T$, $C \to D$ *and* $A \geq [\![C]\!]$*, then there exists* B *such that* $A \to^+ B$ *with* $B \geq [\![D]\!]$.

The details of this proof can again be found in [Dal Lago and Ghyselen 2023], but the result is similar to the one obtained in [Hillerström and Lindley 2018], Theorem 9. From this simulation, we obtain:

Theorem 4. *For any term* $\vdash C : \mathtt{Nat}$ *in PCF, there is a term of finitary HEPCF* $C_f \triangleq \mathtt{let}\ a = [\![C]\!]\ \mathtt{in}\ (a\ ())$ *such that* $\vdash_E C_f : \mathtt{Unit}$ *and* $ET(C_f)$ *represents the normal form of* C *(if it exists, otherwise* $ET(C_f)$ *is* $\bot$*).*

Proof. Suppose that $\vdash C : \mathtt{Nat}$ has no normal form, then $[\![C]\!]$ has no normal form, and thus the effect tree of C_f is $\bot$. Otherwise, suppose that $C \to^* \mathtt{return}(V)$ with V a closed value of type `Nat`. Then, $[\![C]\!] \to^* D$ with $D \geq [\![\mathtt{return}(V)]\!]$. In particular, $D \to^* \mathtt{return}(W)$ with $W \geq [\![V]\!]$. Thus, the effect tree of C_f is the same as $[\![V]\!]$. Because V is a closed value of type `Nat`, this effect tree is the tree representation of the natural number V, which concludes the proof. □

As we did for EPCF, we can define the MSO model-checking problem for HEPCF and easily reach the following result through Theorem 4:

Corollary 2. *The MSO model-checking problem for HEPCF is undecidable.*

Indeed, given a PCF computation of type `Nat`, we can translate it into an HEPCF computation, and ask if the effect tree of this HEPCF computation contains a $\bot$, the latter happening precisely when the initial term halts. Summing up, handlers in their full generality are simply too expressive for MSO model-checking.

8.2 Undecidability for Restricted Output Types

The proof of Theorem 4 strongly depends on the fact that handlers can change the output type, which is essential to encode pairs of computations and create copies of them. Careful readers might wonder whether a fine control over the output type is an essential ingredient of our undecidability proof, and whether we could have a decidable MSO model-checking for constrained handlers

in which the output type if fixed, e.g., $\texttt{Unit}_E \Rightarrow_F \texttt{Unit}$. This is typically the kind of handler we would obtain with an extension of the ECPS language [Matache 2018; Matache and Staton 2019] with handlers. In this section, we give some hints about why the MSO model checking remains undecidabile for such a language, while the complete development can be found in [Dal Lago and Ghyselen 2023].

Let us consider a restriction of the HEPCF language in which all computations *must* have type `Unit`, and no `let` expression is allowed. This may seem like a too severe constraint, but without handlers this is actually the target language of a CPS translation [Matache and Staton 2019] whose source language is EPCF. We use o to denote the unit type here, seen as the answer type for a CPS translation. The important point is that the typing rule for handlers becomes:

$$\frac{F = \{\sigma_i : \mathsf{B}_i \rightsquigarrow k_i \mid 1 \leq i \leq n\} \qquad \Gamma \vdash_E t_r : o \qquad (\Gamma, x : \mathsf{B}_i, r : k \to_E o \vdash_E t_i : o)_{1 \leq i \leq n}}{\Gamma \vdash \{\texttt{return} \mapsto t_r\} \cup \{\sigma_i(x; r) \mapsto t_i \mid 1 \leq i \leq n\} : o_F \Rightarrow_E o}$$

One can easily see that the proof strategy we employed in Section 8.1 is of no help here. We thus have to follow a different path, centered around an encoding of general handlers as restricted handlers . As before, the main question is how to encode a conditional using a fold. The idea we use here is to pick a larger alphabet of algebraic operations: for each operation $\sigma : \mathsf{B} \rightsquigarrow k$ in Σ we add a duplicate of this operation indicated as $\sigma_d : \mathsf{B} \rightsquigarrow k$, and called a *marked* operation. The main idea consists in seeing handlers as effect tree transformers *not* altering the shape of trees, something which even restricted handlers can do. More specifically, we define two handlers:

(1) We first define a handler H_d that takes a computation, and replace any non-marked operation σ by its marked counterpart σ_d, leaving everything else unchanged.
(2) We then define a second handler H_r as follows:

$$H_r \triangleq \{\texttt{return} \mapsto \texttt{return}; \quad \sigma(v; r) \mapsto \sigma(v; y.\texttt{with}\ H_d\ \texttt{handle}\ (r\ y))\}$$

Essentially the handler H_r leaves the first operation it encounters unchanged, while all the subsequent operations are marked through the H_d handler.

Using H_r on a tree t with only unmarked operations, we can somehow distinguish the first operation from all the other ones: it is the only operation that is not marked after the application of H_r. Let us call the resulting tree $H_r(t)$. If we apply to $H_r(t)$ a handler that does nothing on marked operations but that handles non-marked operations, we are implicitly defining a shallow handler on the initial tree t, since we only handle the first operation and do nothing on the subsequent ones. Following this idea, it is possible to encode a conditional for integers, as in Section 8.1, and thus prove undecidability of the restricted language, showing that even without access to output types, handlers remain too expressive for HOMC.

8.3 Recovering Decidability in a Calculus for Generic Effects

Does the result in the last section mean that we have to get away with handlers if HOMC is our concern? Essentially, it rather shows that *general* handlers, define a transformation on effect trees that is simply *too expressive* for HOMC. However, this does not mean that *simpler* tree transformations cannot be expressed as HORS, and this section is devoted to introducing a class of handlers whose underlying tree transformations are indeed amenable to HOMC. Intuitively, an interpretation of an algebraic operation $\sigma : \mathsf{B} \rightsquigarrow k$ could be just a function of type $B \to k$, with no access to the continuation. This is of course much less expressive than fully-fledged handlers, but we will argue that what we obtain is not *too* restrictive. Formally, we can capture all this by a

$$\frac{\Gamma \vdash H : T_E \Rrightarrow_F U \qquad \Gamma \vdash_E C : T}{\Gamma \vdash_F \mathtt{with}\ H\ \mathtt{handle}\ C : U}$$

$$\frac{E = \{\sigma_i : \mathsf{B}_i \rightsquigarrow k_i \mid 1 \le i \le n\} \qquad \Gamma, x : T \vdash_F C_r : U \qquad (\Gamma, x : \mathsf{B}_i, \vdash_F C_i : k_i)_{1 \le i \le n}}{\Gamma \vdash \{\mathtt{return}(x) \mapsto C_r\} \cup \{\sigma_i(x) \mapsto C_i \mid 1 \le i \le n\} : T_E \Rrightarrow_F U}$$

$$\frac{C \to D}{\mathtt{with}\ H\ \mathtt{handle}\ C \to \mathtt{with}\ H\ \mathtt{handle}\ D}$$

$$\frac{H = \{\mathtt{return}(x) \mapsto C_r\} \cup \{\sigma_i(x) \mapsto C_i \mid 1 \le i \le n\}}{\mathtt{with}\ H\ \mathtt{handle}\ \mathtt{return}(V) \to C_r[x := V]}$$

$$\frac{H = \{\mathtt{return}(x) \mapsto C_r\} \cup \{\sigma_i(x) \mapsto C_i \mid 1 \le i \le n\}}{\mathtt{with}\ H\ \mathtt{handle}\ \sigma_i(V; y.C) \to \mathtt{let}\ y = C_i[x := V]\ \mathtt{in}\ (\mathtt{with}\ H\ \mathtt{handle}\ C)}$$

Fig. 9. Static and Dynamic Semantics of GEPCF

restriction of handlers, defined in the following language, called GEPCF (for Generic Effects PCF):

$$\begin{array}{lll}
\text{(Types)} & T, U ::= \mathsf{B} \mid T \to_E U \mid k \\
\text{(Effects)} & E ::= \emptyset \mid \{\sigma : \mathsf{B} \rightsquigarrow k\} \cup E \\
\text{(Values)} & V, W ::= v \mid \underline{n} \mid x \mid \lambda x.C \mid \mathtt{fix}\ f.V \\
\text{(Computations)} & C, D ::= V\ W \mid \sigma(V; x.C) \mid \mathtt{return}(V) \mid \mathtt{let}\ x = C\ \mathtt{in}\ D \\
 & \qquad \mid \mathtt{with}\ H\ \mathtt{handle}\ C \mid \mathtt{case}(V; C_1, \ldots, C_k) \\
\text{(Handlers)} & H ::= \{\mathtt{return}(x) \mapsto C_r\} \cup \{\sigma_i(x) \mapsto C_i \mid 1 \le i \le n\}
\end{array}$$

Note that as expected, the handler for an algebraic operation does not have access to the continuation. The static and dynamic semantics are in Figure 9.

Please observe that we can always simulate a GEPCF handler by a HEPCF handler:

$$\mathtt{return} \mapsto C_r; \qquad \sigma_i(x; r) \mapsto \mathtt{let}\ z = C_i\ \mathtt{in}\ (r\ z).$$

Thus, generic handlers are a specific case of standard handlers in which the handling of an operation σ_i always has the shape $\mathtt{let}\ z = C_i\ \mathtt{in}\ (r\ z)$. Informally, if we look at it from the point of view of tree transformations, this is exactly what we would obtain from a generic effect for the effect tree monad, that is why we call this language GEPCF. Indeed, by definition [Plotkin and Power 2003], a generic effect of an algebraic operation $\sigma_i : B_i \rightsquigarrow k_i$ for the effect tree monad is given by a Kleisli map in $B_i \to ET(k_i)$. This is precisely the type of C_i in the restricted handler. The reduction relation then implements the usual 1-1 correspondence between generic effects and natural algebraic operations, which here corresponds to using tree composition: the effect tree produced by $\mathtt{let}\ y = C_i[x := V]\ \mathtt{in}\ (\mathtt{with}\ H\ \mathtt{handle}\ C)$ is the same as the effect tree produced by $C_i[x := V]$ except that all $\mathtt{return}(\underline{n})$ leaves are replaced by the tree of $(\mathtt{with}\ H\ \mathtt{handle}\ C[y := \underline{n}])$.

Let us now show that this language can be translated back into the λY-calculus, thus proving MSO-decidability.[1]

Definition 16 (Translation from GEPCF to λ-Y calculus). *We define the translation $(\cdot)^\star$ on types and terms. For the sake of conciseness, we slightly abuse notation about pairs in the left-hand*

[1] An alternative proof could consist in showing that the tree transformation induced by this handler is a special case of a tree transducer as in [Kobayashi et al. 2010]: a handler in GEPCF behaves as a unique case operator of [Kobayashi et al. 2010] repeated until a leaf is reached.

side of an arrow type:

$$\mathsf{B}^\star \triangleq o; \qquad k^\star \triangleq o^k \to o; \qquad (T \to_E U)^\star \triangleq T^\star \to E^\star \to \neg U^\star \to o;$$

$$\emptyset^\star \triangleq (); \qquad E \cup \sigma : \mathsf{B} \rightsquigarrow k^\star \triangleq (E^\star, o \to \neg k^\star \to o).$$

We want to show that if $\Gamma \vdash V : T$ *then* $\Gamma^\star \vdash V^\star : T^\star$ *and that if* $\Gamma \vdash_E C : T$ *then* $\Gamma^\star \vdash C^\star : E^\star \to (T^\star \to o) \to o$. *Informally, we go via a CPS-translation in which a* handler continuation, *of type* $E^\star$ *is carried around, giving an interpretation to all algebraic operations in* E. *We take the* λY*-calculus with a signature* Σ *including at least all the constants values of EPCF. Formally, the translation acts on computations and values as follows, where* H *stands for* $\{\mathtt{return}(x) \mapsto C_r; \sigma_i(x) \mapsto C_i \mid 1 \leq i \leq n\}$*:*

$$v^\star \triangleq v \qquad \underline{n}^\star \triangleq \lambda(x_1, \ldots, x_k).x_n$$

$$x^\star \triangleq x \qquad (\lambda x.C)^\star \triangleq \lambda x.C^\star$$

$$(\mathtt{fix}\ x.V)^\star \triangleq Y\ (\lambda x.V^\star) \qquad (V\ W)^\star \triangleq V^\star\ W^\star$$

$$(\mathtt{return}(V))^\star \triangleq \lambda(\widetilde{h}, c).c\ V^\star \qquad (\sigma_i(V; x.C))^\star \triangleq \lambda(\widetilde{h}, c).h_i\ V^\star\ (\lambda x.C^\star\ \widetilde{h}\ c)$$

$$(\mathtt{let}\ x = C\ \mathtt{in}\ D)^\star \triangleq \lambda(\widetilde{h}, c).C^\star\ \widetilde{h}\ (\lambda x.D^\star\ \widetilde{h}\ c)$$

$$(\mathtt{case}(V; C_1, \ldots, C_k))^\star \triangleq \lambda(\widetilde{h}, c).V^\star\ (C_1^\star\ \widetilde{h}\ c)\ \cdots\ (C_k^\star\ \widetilde{h}\ c)$$

$$(\mathtt{with}\ H\ \mathtt{handle}\ C)^\star \triangleq \lambda(\widetilde{h}, c).C^\star\ (\lambda(x, r).C_1^\star\ \widetilde{h}\ r)\ \cdots\ (\lambda(x, r).C_n^\star\ \widetilde{h}\ r)\ (\lambda x.C_r^\star\ \widetilde{h}\ c).$$

Lemma 6. *If* $\Gamma \vdash V : T$ *then* $\Gamma^\star \vdash V^\star : T^\star$, *while if* $\Gamma \vdash_E C : T$ *then* $\Gamma^\star \vdash C^\star : E^\star \to \neg\neg T^\star$

The proof is straightforward.

Lemma 7. *If* $\Gamma \vdash_E C : T$ *and* $C \to D$ *then for any well-typed continuations* $(\widetilde{h}, c)$, *we have* $C^\star_{\widetilde{h},c} \to^+ B =_{adm} D^\star_{\widetilde{h},c}$, *where* $=_{adm}$ *is the least congruence relation including all pairs in the form* $((\lambda(\widetilde{h}, c).C)\ \widetilde{h}\ c, C)$.

Proof. The proof goes by induction on the derivation of $C \to D$. Some cases are equivalent to those of the usual CPS translation so we can safely ignore them. Similarly to Lemma 2, we have to take care of some administrative reduction steps. The interesting cases are those dealing with handlers. The contextual case is straightforward since we can reduce the head of the λY-term. The most interesting case is the one about the handling of an algebraic operation. For this case, we have that:

$$(\mathtt{with}\ H\ \mathtt{handle}\ \sigma_i(V; y.C))^\star\ \widetilde{h}\ c \to^+ (C_i^\star)[x := V^\star]\ \widetilde{h}\ (\lambda y.C^\star\ \widetilde{\mathcal{H}}\ C),$$

where $\widetilde{\mathcal{H}}$ and C denotes continuations such that

$$(\mathtt{with}\ H\ \mathtt{handle}\ C)^\star \triangleq \lambda(\widetilde{h}, c).C^\star\ \widetilde{\mathcal{H}}\ C.$$

On the other hand, we have that:

$$(\mathtt{let}\ y = C_i[x := V]\ \mathtt{in}\ (\mathtt{with}\ H\ \mathtt{handle}\ C))^\star\ \widetilde{h}\ c \to_{adm} C_i^\star[x := V]\ \widetilde{h}\ (\lambda y.(\lambda(\widetilde{h}, c).C^\star \widetilde{\mathcal{H}}\ C)\ \widetilde{h}\ c),$$

and those two terms are administratively equivalent. □

We can now define a canonical continuation handler for a given set of effects.

Definition 17. *Consider a set* $E = \{\sigma_i : \mathsf{B}_i \rightsquigarrow k_i\}$ *of effects. We pose* $\Sigma_E = \{\sigma_i : k_i + 1\}$ *a signature for terms of the* λY*-calculus. We define the identity continuation handler for* E *as the* λY*-term of type* $E^\star$*:*

$$(\widetilde{h}_E)_i \triangleq \lambda(x, f).\sigma_i\ x\ (f\ (\lambda x_1, \ldots, x_k.x_1))\ \ldots\ (f\ (\lambda x_1, \ldots, x_k.x_k))$$

In particular, remark that $(\sigma_i(V;x.C))^\star_{\widetilde{h}_E,c}$ *becomes:*

$$(\sigma_i(V;x.C))^\star_{\widetilde{h}_E,c} \to \sigma_i\ V^\star\ (C^\star[x := \underline{1}^\star]\ \widetilde{h}_E\ c)\ \ldots\ (C^\star[x := \underline{k}^\star]\widetilde{h}_E\ c)$$

which corresponds exactly to the effect tree of $\sigma_i(V;x.C)$.

We are now ready to prove the main result of this section:

Theorem 5. *For any* $\vdash_E C : T$, *for any continuation* $\vdash c : T^\star \to o$, *we have that*

$$ET(C)[\mathtt{return}(V) \leftarrow BT(c\ V^\star)] = BT(V^\star\ \widetilde{h}_E\ c)$$

Proof. We can proceed by induction on the effect tree of C:

- If C can be reduced indefinitely, then by simulation, $C^\star\ \widetilde{h}_E\ c$ can be reduced indefinitely too.
- If $C \to^* \mathtt{return}(V)$. Then $C^\star\ \widetilde{h}_E\ c \to^* c\ V^\star$ by definition of $\mathtt{return}(V)^\star$.
- If $C \to^* \sigma(V;x.D)$. Then $C^\star\ \widetilde{h}_E\ c \to^* \sigma_i\ V^\star\ (D^\star[x := \underline{1}^\star]\ \widetilde{h}_E\ c)\ \cdots\ (D^\star[x := \underline{k}^\star]\widetilde{h}_E\ c)$. And we can conclude by induction hypothesis.

□

In particular, if we have T as a base type and a continuation that is the encoding of the identity, we can recover exactly the initial effect tree. Moreover, we can naturally define the MSO model-checking problem for GEPCF, as we did for EPCF and HEPCF, and obtain that:

Corollary 3. *The MSO model-checking prolem for GEPCF is decidable.*

Please note that, similarly to what happens in Corollary 1, the continuation c can be chosen arbitrarily.

9 RELATED WORK

Effectful Higher-Order Programs. This is definitely not the first paper concerned with higher-order effectful programs. The denotational semantics of calculi having this nature has been studied since Moggi's seminal work on monads [Moggi 1988], implicitly providing notions of equivalence and refinement. All this has been given a more operational flavor in Plotkin and Power's account on adequacy for algebraic effects [Plotkin and Power 2003], from which the operational semantics presented in this paper is greatly inspired. Logics for algebraic effects have been introduced by Pretnar and Plotkin [Plotkin and Pretnar 2008], by Matache and Staton [Matache and Staton 2019], and by Simpson and Voorneveld [Simpson and Voorneveld 2019]. The latter has certainly been another major source of inspiration although, as explicitly stated by the authors[2], automatic verification techniques were considered simply as out of scope. In fact, we are not aware of any attempt to study the decidability of the aforementioned theories.

Verification of Infinite State Higher-Order Program with Control Operators and Effects. A recent line of work has been concerned with the temporal verification of infinite-state higher-order programs with control operators using type and effect systems. For example, Gordon [Gordon 2020] defines a framework for sequential effects with tagged control operators akin to `abort` and `call/cc`, capturing temporal safety properties. Similarly, [Sekiyama and Unno 2023] describes a verification methodology for general temporal properties in presence of the control operators `shift0` and `reset0`, while [Song et al. 2022] tackles the problem of verifying general effect handlers against

[2]In [Simpson and Voorneveld 2019], Section 10, Paragraph 8: "We view the infinitary propositional logic of this paper as providing a low-level language, into which practical high-level finitary logics for expressing program properties can potentially be compiled. [...] We view the development of such high-level logics and their compositional reasoning principles, aimed at practical specification and verification, as a particularly promising topic for future research."

specifications in a logic more expressive than classical LTL. However, we are not aware of any work dealing with the problem of verifying general effect handlers against MSO formulas, nor of any undecidability results for the problem.

Higher-Order Model Checking. Model checking of higher-order programs has been an active topic of investigation in the last twenty years, with many positive results, from the pioneering and partial results by Knapik et al. [Knapik et al. 2001, 2002] to Ong's already mentioned breakthrough result [Ong 2006], followed by Kobayashi and co-authors' work on model checking as (intersection) type checking [Kobayashi and Ong 2009]. Noticeably, some of these works go in the direction of extending the aforementioned decidability results to higher-order calculi endowed with some *specific* form of effect, like probabilistic choice [Kobayashi et al. 2020], some form of resource usage [Kobayashi 2009] or exception-handling [Sato et al. 2013]. Outcomes are not always been on the positive side, as undecidability of the model checking problem for probabilistic variations on HORSs shows. Again, we are not aware of any study aimed at giving general criteria for decidability.

Effect Handlers. Effect handlers are a linguistic feature allowing to give computational meaning to algebraic operations through handlers, i.e. routines specifically designed for the handling of these effects, which in this way can be managed internally by the program itself rather than by the environment. Given their elegance and naturalness in generalizing standard language constructions like the `try with` operator for exceptions, handlers have been largely studied both theoretically and concretely [Bauer 2019; Biernacki et al. 2019; Hillerström and Lindley 2018; Hillerström et al. 2017; Kammar et al. 2013; Plotkin and Pretnar 2009; Sekiyama et al. 2020]. We are not aware of any study about handlers in a finitary setting, and even less about questions of decidability with regards to higher-order model checking.

10 CONCLUSION

This paper tackles, for the first time in a general way, the problem of evaluating the intrinsic difficulty of the higher-order model checking problem when applied to programs that exhibit effects, possibly managed through handlers. The results obtained are in two styles: while the problem of *capturing* algebraic operations in calculi amenable to HOMC does not pose problems and indeed can be solved in its generality, *observing* the effects produced by such operations and *handling* them must be done with great care: we observe that in general this leads to undecidability, but that in both cases, criteria can be defined allowing to keep the problem decidable. This consists, respectively, in observing the effects so that this observation can be expressed in MSO and in a restricted class of handlers sufficient for the modeling of the so-called generic effects.

There are aspects that this paper deliberately overlooks, but which certainly deserve further study. First of all, it should be mentioned that the impact of effects on (known) complexity results about HOMC is not studied in detail here, but that the introduced translations (in particular those in Section 5 and Section 8) could perhaps be implemented more efficiently, following works on order optimization [Nakamura et al. 2020]. It should also be said that the possibility of introducing logics more powerful than MSO this way capturing quantitative observations without possibly falling back into the known cases of undecidability (e.g., in the case of probabilistic effects) is a very interesting research direction that the authors intend to investigate in the immediate future.

DATA AVAILABILITY STATEMENT

The long version of this paper can be found on arXiv:2308.16542 [cs.LO] [Dal Lago and Ghyselen 2023].

REFERENCES

Andrej Bauer. 2019. What is Algebraic About Algebraic Effects and Handlers? arXiv:1807.05923 [cs.LO]

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting Algebraic Effects. *Proceedings of the ACM for Programming Languages* 3, (POPL) (2019), 6:1–6:28.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593.

Christopher Broadbent, Arnaud Carayol, Luke Ong, and Olivier Serre. 2010. Recursion Schemes and Logical Reflection. In *Proc. of LICS 2010*. IEEE, 120–129.

Christopher Broadbent and Naoki Kobayashi. 2013. Saturation-Based Model Checking of Higher-Order Recursion Schemes. In *Proc. of CSL 2013 (LIPIcs, Vol. 23)*. 129–148.

Arnaud Carayol and Olivier Serre. 2012. Collapsible Pushdown Automata and Labeled Recursion Schemes: Equivalence, Safety and Effective Selection. In *Proc. of LICS 2012*. IEEE, 165–174.

Pierre Clairambault and Andrzej S. Murawski. 2013. Böhm Trees as Higher-Order Recursive Schemes. In *Proc. of FSTTCS 2013 (LIPIcs, Vol. 24)*. 91–102.

Edmund M. Clarke. 1997. Model checking. In *Proc. of FSTTCS 1997*. Springer, 54–56.

Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, Roderick Bloem, et al. 2018. *Handbook of model checking*. Vol. 10. Springer.

Bruno Courcelle. 1995. The monadic second-order logic of graphs IX: Machines and their behaviours. *Theoretical Computer Science* 151, 1 (1995), 125–162.

Ugo Dal Lago, Francesco Gavazzo, and Paul Levy. 2017. Effectful applicative bisimilarity: Monads, relators, and Howe's method. In *Proc. of LICS 2017*. IEEE, 1–12.

Ugo Dal Lago and Alexis Ghyselen. 2023. On Model-Checking Higher-Order Effectful Programs (Long Version). arXiv:2308.16542 [cs.LO]

Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In *Proc. of ICFP 2000*. ACM, 198–208.

Philippe de Groote. 1994. A CPS-translation of the $\lambda\mu$-calculus. In *Proc. of CAAP 1994*. Springer, 85–99.

E. Allen Emerson and Charanjit S. Jutla. 1991. Tree automata, mu-calculus and determinacy. In *Proc. of FOCS 1991*. IEEE, 368–377.

Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proc. of PLDI 1991*. ACM, 268–277.

Colin S. Gordon. 2020. Lifting Sequential Effects to Control Operators. In *Proc. of ECOOP 2020 (LIPIcs, Vol. 166)*. 23:1–23:30.

Erich Grädel, Wolfgang Thomas, and Thomas Wilke. 2003. *Automata, logics, and infinite games: a guide to current research*. LNCS, Vol. 2500. Springer.

Matthew Hague, Andrzej Murawski, Luke Ong, and Olivier Serre. 2008. Collapsible Pushdown Automata and Recursion Schemes. In *Proc. of LICS 2008*. IEEE, 452–461.

Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *Proc. of ESOP 2018*. Springer, 415–435.

Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *Proc. of FSCD 2017 (LIPIcs, Vol. 84)*. 18:1–18:19.

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the correctness of reactive systems using sized types. In *Proc. of POPL 1996*. ACM, 410–423.

Patricia Johann, Alex Simpson, and Janis Voigtländer. 2010. A Generic Operational Metatheory for Algebraic Effects. In *Proc. of LICS 2010*. IEEE, 209–218.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proc. of ICFP 2013*. ACM, 145–158.

James King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

Teodor Knapik, Damian Niwiński, and Paweł Urzyczyn. 2001. Deciding Monadic Theories of Hyperalgebraic Trees. In *Proc. of TLCA 2001*. Springer, 253–267.

Teodor Knapik, Damian Niwiński, and Paweł Urzyczyn. 2002. Higher-Order Pushdown Trees Are Easy. In *Proc. of FoSSaCS 2002*. Springer, 205–222.

Naoki Kobayashi. 2009. Types and higher-order recursion schemes for verification of higher-order programs. In *Proc. of POPL 2019*. ACM, 416–428.

Naoki Kobayashi. 2011. A Practical Linear Time Algorithm for Trivial Automata Model Checking of Higher-Order Recursion Schemes. In *Proc. of FoSSaCS 2011*. Springer, 260–274.

Naoki Kobayashi, Ugo Dal Lago, and Charles Grellois. 2020. On the termination problem for probabilistic higher-order recursive programs. *Logical Methods in Computer Science* 16, 4 (2020).

Naoki Kobayashi and Atsushi Igarashi. 2013. Model-Checking Higher-Order Programs with Recursive Types. In *Proc. of ESOP 2013*. Springer, 431–450.

Naoki Kobayashi and C-H Luke Ong. 2009. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proc. of LICS 2009*. IEEE, 179–188.

Naoki Kobayashi, Naoshi Tabuchi, and Hiroshi Unno. 2010. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Proc. of POPL 2010*. ACM, 495–508.

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (jul 2009), 107–115.

Cristina Matache. 2018. Program equivalence for algebraic effects via modalities.. In *Master's thesis*. University of Oxford, Department of Computer Science.

Cristina Matache and Sam Staton. 2019. A Sound and Complete Logic for Algebraic Effects. In *Proc. of FoSSaCS 2019*. Springer, 382–399.

Eugenio Moggi. 1988. *Computational Lambda-calculus and Monads*. University of Edinburgh, Department of Computer Science.

Yoshiki Nakamura, Kazuyuki Asada, Naoki Kobayashi, Ryoma Sin'ya, and Takeshi Tsukada. 2020. On Average-Case Hardness of Higher-Order Model Checking. In *Proc. of FSCD 2020 (LIPIcs, Vol. 167)*. 21:1–21:23.

Robin Neatherway, Steven Ramsay, and Luke Ong. 2012. A Traversal-Based Algorithm for Higher-Order Model Checking. In *Proc. of ICFP 2012*. ACM, 353–364.

Luke Ong. 2006. On model-checking trees generated by higher-order recursion schemes. In *Proc. of LICS 2006*. IEEE, 81–90.

Luke Ong. 2015. Higher-order model checking: An overview. In *Proc. of LICS 2015*. IEEE, 1–15.

Gordon Plotkin. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science* 1, 2 (1975), 125–159.

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

Gordon Plotkin and Matija Pretnar. 2008. A logic for algebraic effects. In *Proc. of LICS 2008*. IEEE, 118–129.

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Proc. of ESOP 2009*. Springer, 80–94.

Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. In *Proc. of MFPS 2015 (Electronic Notes in Theoretical Computer Science, Vol. 319)*. Elsevier, 19–35.

Steven Ramsay, Robin Neatherway, and Luke Ong. 2014. A Type-Directed Abstraction Refinement Approach to Higher-Order Model Checking. In *Proc. of POPL 2014*. ACM, 61–72.

Sylvain Salvati and Igor Walukiewicz. 2012. Recursive Schemes, Krivine Machines, and Collapsible Pushdown Automata. In *Proc. of RP 2012*. Springer, 6–20.

Sylvain Salvati and Igor Walukiewicz. 2014. Krivine machines and higher-order schemes. *Information and Computation* 239 (2014), 340–355.

Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. 2013. Towards a Scalable Software Model Checker for Higher-Order Programs. In *Proc. of PEPM 2013*. ACM, 53–62.

Taro Sekiyama, Takeshi Tsukada, and Atsushi Igarashi. 2020. Signature Restriction for Polymorphic Algebraic Effects. *Proceedings of the ACM on Programming Languages* 4, (ICFP) (2020), 117:1–117:30.

Taro Sekiyama and Hiroshi Unno. 2023. Temporal Verification with Answer-Effect Modification: Dependent Temporal Type-and-Effect System with Delimited Continuations. *Proceedings of the ACM on Programming Languages* 7, (POPL) (2023), 2079–2110.

Alex Simpson and Niels Voorneveld. 2019. Behavioural Equivalence via Modalities for Algebraic Effects. *ACM Trans. Program. Lang. Syst.* 42 (2019), 4:1–4:45.

Yahui Song, Darius Foo, and Wei-Ngan Chin. 2022. Automated Temporal Verification For Algebraic Effects. In *Proc. of APLAS 2022*. Springer, 88–109.

Sam Tobin-Hochstadt and David Van Horn. 2012. Higher-order symbolic execution via contracts. In *Proc. of OOPSLA 2012*. ACM, 537–554.

Takeshi Tsukada and Naoki Kobayashi. 2014. Complexity of Model-Checking Call-by-Value Programs. In *Proc. of FoSSaCS 2014*. Springer, 180–194.

Igor Walukiewicz. 1993. On completeness of the mu-calculus. In *Proc. of LICS 1993*. IEEE, 136–146.

Igor Walukiewicz. 2016. Automata Theory and Higher-Order Model-Checking. *ACM SIGLOG News* 3, 4 (2016), 13–31.

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