Reachability is Decidable for ATM-Typable Finitary PCF with Effect Handlers

Ryunosuke Endo[1] and Tachio Terauchi[2]

[1] Waseda University, Tokyo, Japan**
minerva@ruri.waseda.jp
[2] Waseda University, Tokyo, Japan
terauchi@waseda.jp

Abstract. It is well known that the reachability problem for simply-typed lambda calculus with recursive definitions and finite base-type values (finitary PCF) is decidable. A recent paper by Dal Lago and Ghyselen has shown that the same problem becomes undecidable when the language is extended with algebraic effect and handlers (effect handlers). We show that, perhaps surprisingly, the problem becomes decidable even with effect handlers when the type system is extended with answer type modification (ATM). A natural intuition may find the result contradictory, because one would expect allowing ATM makes more programs typable. Indeed, this intuition is correct in that there are programs that are typable with ATM but not without it, as we shall show in the paper. However, a corollary of our decidability result is that the converse is true as well: there are programs that are typable without ATM but becomes untypable with ATM, and we will show concrete examples of such programs in the paper. Our decidability result is proven by a novel continuation passing style (CPS) transformation that transforms an ATM-typable finitary PCF program with effect handlers to a finitary PCF program without effect handlers. Additionally, as another application of our CPS transformation, we show that every recursive-function-free ATM-typable finitary PCF program with effect handlers terminates, while there are (necessarily ATM-untypable) recursive-function-free finitary PCF programs with effect handlers that may diverge. Finally, we disprove a claim made in a recent work that proved a similar but strictly weaker decidability result. We foresee our decidability result to lay a foundation for developing verification methods for programs with effect handlers, just as the decidability result for reachability of finitary PCF has done such for programs without effect handlers.

1 Introduction

A popular approach to the verification of infinite-state programs is to abstract the programs so that their base-type values are over finite domains, as seen in, for example, predicate abstraction with CEGAR [4,7,3,39,28,18,42,36,34,9,8,19,26].

** Author's current affiliation: Mizuho Bank, Ltd.

Importantly, the reachability problem, which asks whether there exists an execution of the program reaching a certain program state (typically a designated "error" state), is known to be decidable for such programs when they are simply-typable, even when the programs contain higher-order and recursive functions. That is, reachability for finitary PCF is decidable [27,16,41].[1]

Meanwhile, algebraic effects and handlers (*effect handlers* henceforth) are a programming language feature for expressing computational effects such as mutable state, exception, and non-determinism [35]. They have a theoretical origin, stemming from the research on denotational semantics [29,30,32,31,33,5], but are also practically popular and have been incorporated into many popular programming languages such as C, C++, Java, OCaml, and Haskell [21,6,45,38,12,1]. Unfortunately, a recent paper [20] has shown that reachability, which is decidable for finitary PCF as mentioned above, becomes undecidable when the language is extended with effect handlers.

In this paper, we show that this undecidability stems from the way the standard type systems for effect handlers are designed. More concretely, we show that, perhaps surprisingly, extending the standard type system to allow answer type modification (ATM) [10,2,15] recovers decidability. A natural intuition may find the result contradictory because one would expect allowing ATM makes more programs typable. Indeed, this intuition is correct in that there are programs that are typable with ATM but not without it, as we shall show in the paper (cf. Example 5). However, a corollary of our decidability result is that the converse is true as well: there are programs that are typable without ATM but becomes untypable with ATM, and we will show concrete examples of such programs in the paper (cf. Sections 3.2 and 3.3).

Our decidability result is proven by a novel continuation passing style (CPS) transformation that transforms an ATM-typable finitary PCF program with effect handlers to a finitary PCF program without effect handlers. Then, our decidability result follows from the fact that the target of the CPS transformation, finitary PCF, has decidable reachability as mentioned in the first paragraph of this section.

Additionally, as another application of our CPS transformation, we show that every recursive-function-free ATM-typable finitary PCF program with effect handlers terminates, while there are (necessarily ATM-untypable) recursive-function-free finitary PCF programs with effect handlers that may diverge. Finally, we disprove in a claim made in a recent work that proved a similar but strictly weaker decidability result [37]. In summary, the main contributions of the paper are as follows:

- We show that reachability for ATM-typable finitary PCF with effect handlers is decidable. A novel CPS transformation is introduced to prove the result.

[1] The result can be seen as an extension of the decidability result for reachability of pushdown systems [11], which correspond to first-order recursive programs, to higher-order recursive programs. Also, the result should not be confused with the result that observational equivalence for finitary PCF is undecidable [23].

$v ::= x \mid () \mid \top \mid \bot \mid \lambda x.c \mid \texttt{rec}\ x = v$
$c ::= \texttt{return}\ v \mid op\ v \mid v_1\ v_2 \mid \texttt{if}\ v\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{let}\ x = c_1\ \texttt{in}\ c_2 \mid \texttt{with}\ h\ \texttt{handle}\ c$
$h ::= \{\texttt{return}\ x = c, \overline{op_i\ x_i\ k_i = c_i}\}$

Fig. 1. The syntax of FPCF$_{\mathit{eff}}$.

- A corollary of our decidability result is that there are finitary PCF programs typable without ATM but untypable with it, and we show concrete examples of such programs.
- As another application of the CPS transformation, we show that recursive-function-free ATM-typable finitary PCF programs with effect handlers always terminate, while there are (ATM-untypable) recursive-function-free finitary PCF programs with effect handlers that may diverge.
- We disprove a claim made in a recent paper [37] regarding what the paper calls the number of "active effect handlers".

The rest of the paper is organized as follows. Section 2 defines preliminary notions. Section 3 contains the main results mentioned above. Section 4 discusses related work. Section 5 concludes the paper. For space, some materials are deferred to the appendix.

2 Preliminaries

Figure 1 shows the syntax of untyped finitary PCF with effect handlers FPCF$_{\mathit{eff}}$. As in many presentations of effect handlers [35,15], we adopt the approach of call-by-push-value λ calculus [22] to separate the syntax of expressions into *values*, ranged over by meta variables v, and *computations*, ranged over by c. A *handler*, ranged over by h, consists of a single *return clause* of the form $\texttt{return}\ x = c$ and finitely many *operation clauses* of the form $op_i\ x_i\ k_i = c_i$. The parameter k_i in an operation clause is called the *continuation* parameter. Recursive definitions are given by the syntax $\texttt{rec}\ x = v$ which recursively binds x in v.

Note that functions, arguments, and conditional expressions are restricted to values, but this does not reduce expressivity because, for example, a function application $c\ v$ can be expressed as $\texttt{let}\ x = c\ \texttt{in}\ x\ v$ using a fresh variable x. For convenience, we often assume this convention and allow computations to appear at positions where values are expected. For example, we may write $x\ y\ z$ for $\texttt{let}\ w = x\ y\ \texttt{in}\ w\ z$, adopting the usual convention that function application is left associative. Conversely, when a value v appears at a position where a computation in expected, we read it as $\texttt{return}\ v$. For example, we may write $\lambda x.\lambda y.x$ for $\lambda x.\texttt{return}\ \lambda y.\texttt{return}\ x$. We also write $c_1; c_2$ for $\texttt{let}\ x = c_1\ \texttt{in}\ c_2$ where x is not free in c_2. As usual, a *program* is a closed expression.

For concreteness, FPCF$_{\mathit{eff}}$ uses Booleans and unit as base-type values, but our results can be easily be adopted to other finite base-type domains such as variant types and integers modulo a constant. Note that FPCF$_{\mathit{eff}}$ is untyped.

$$\frac{c_1 \to c_2}{\texttt{let}\ x = c_1\ \texttt{in}\ c \to \texttt{let}\ x = c_2\ \texttt{in}\ c}\ \text{(E-Let)} \qquad \frac{}{\texttt{let}\ x = \texttt{return}\ v\ \texttt{in}\ c \to c[v/x]}\ \text{(E-Ret)}$$

$$\frac{}{(\lambda x.c)\ v \to c[v/x]}\ \text{(E-LamApp)} \qquad \frac{}{(\texttt{rec}\ x = v)\ v' \to v[(\texttt{rec}\ x = v)/x]\ v'}\ \text{(E-RecApp)}$$

$$\frac{}{\texttt{if}\ \top\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \to c_1}\ \text{(E-IfTrue)} \qquad \frac{}{\texttt{if}\ \bot\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \to c_2}\ \text{(E-IfFalse)}$$

$$\frac{c \to c'}{\texttt{with}\ h\ \texttt{handle}\ c \to \texttt{with}\ h\ \texttt{handle}\ c'}\ \text{(E-Han)} \qquad \frac{\texttt{return}\ x = c \in h}{\texttt{with}\ h\ \texttt{handle}\ \texttt{return}\ v \to c[v/x]}\ \text{(E-HRet)}$$

$$\frac{op\ x\ k = c \in h}{\texttt{with}\ h\ \texttt{handle}\ E[op\ v] \to c[v/x, \lambda y.\texttt{with}\ h\ \texttt{handle}\ E[\texttt{return}\ y]/k]}\ \text{(E-Op)}$$

Fig. 2. The semantics of $\text{FPCF}_{\mathit{eff}}$.

Later in the paper, we present two type systems for it, an ordinary simple type system $\vdash_{\text{st}}$ and a type system with answer-type modification $\vdash_{\text{atm}}$, and investigate how they affect the decidability of reachability.

Figure 2 shows the operational semantics of $\text{FPCF}_{\mathit{eff}}$. Here, the *evaluation context* E is defined by: $E ::= [] \mid \texttt{let}\ x = E\ \texttt{in}\ c$. The semantics is standard for a language with effect handlers. A key rule is (E-Op) which invokes an operation. An operation invocation is quite different from an ordinary function call, and replaces the current context up to and including the nearest with-handle block with the body of the operation clause c. The actual argument v gets bound to the formal parameter x and the continuation parameter k gets the captured *delimited continuation* $\lambda y.\texttt{with}\ h\ \texttt{handle}\ E[\texttt{return}\ y]$. The behavior is similar to that of the *shift* operator from delimited control [10,2]. Note that the evaluation context E in the delimited continuation is wrapped in the with-handle block, following the standard *deep-handler* semantics [14]. Another important rule is (E-HRet) which processes a return clause invocation. Note that unlike the ordinary return of (E-Ret), a return at a tail position of a with-handle block invokes the return clause so that the returned result is (the evaluation result of) $c[v/x]$ rather than v. As standard, we write $\to^*$ to denote the reflexive transitive closure of $\to$. The *reachability problem* is defined as follows.

Definition 1 (Reachability). The *reachability problem* is to decide, given a program c, if $c \to^* \texttt{return}\ \top$.

We note that the choice of $\top$ is arbitrary. We could have alternatively chosen any other base-type value as the final value that we would like to decide if the program returns or not.

Example 1. Let c_{ex1} be the following program.[2]

$$\begin{array}{l}(\texttt{with}\ h_{st}\ \texttt{handle}\ (\texttt{rec}\ \mathit{loop} = \lambda\_.\texttt{let}\ n = \mathit{get}\ ()\ \texttt{in} \\ \qquad\qquad\qquad\qquad (\texttt{if}\ v_{fst}\ n\ \texttt{then if}\ v_{snd}\ n\ \texttt{then}\ c_{incr}; \mathit{loop}\ ()\ \texttt{else}\ () \\ \qquad\qquad\qquad\qquad \texttt{else}\ c_{incr}; \mathit{loop}\ ()))\ (v_{tup}\ \bot\ \bot)); \texttt{return}\ \top\end{array}$$

where

$$\begin{array}{rl} v_{tup} & \triangleq \lambda x.\lambda y.\lambda f.f\ x\ y \quad v_{fst} \triangleq \lambda p.p\ \lambda x.\lambda y.x \quad v_{snd} \triangleq \lambda p.p\ \lambda x.\lambda y.y \\ c_{inc} & \triangleq \texttt{let}\ n = \mathit{get}\ ()\ \texttt{in} \\ & \quad \texttt{if}\ v_{fst}\ n\ \texttt{then if}\ v_{snd}\ n\ \texttt{then}\ ()\ \texttt{else}\ \mathit{set}\ (v_{tup}\ \bot\ \top) \\ & \quad \texttt{else if}\ v_{snd}\ n\ \texttt{then}\ \mathit{set}\ (v_{tup}\ \top\ \top)\ \texttt{else}\ \mathit{set}\ (v_{tup}\ \top\ \bot) \\ h_{st} & \triangleq \{\texttt{return}\ x = \lambda s.x, \mathit{set}\ x\ k = \lambda s.k\ ()\ x, \mathit{get}\ x\ k = \lambda s.k\ s\ s\} \end{array}$$

The handler h_{st} adopts the standard state-passing approach for expressing mutable states with effect handlers [35]. Namely, the operation *set* updates the current state with the given argument and the operation *get* returns the current state. In this program, a state is a pair of Booleans encoding a 2-bit positive integer. We use the standard λ calculus encoding of pairs: v_{tup}, v_{fst}, and v_{snd} respectively creates a pair, projects the first element, and the projects the second element. The computation c_{inc} is effectful and uses *get* and *set* to increment the current state by one. The initial state is set to be zero (i.e., the pair $(\bot, \bot)$), and the recursive function defined by $\texttt{rec}\ \mathit{loop} = \ldots$ repeatedly increments the state until the value becomes one (i.e., $(\top, \bot)$). Because the initial state is zero, one is eventually reached and the program c_{ex1} terminates returning $\top$. The same program would also terminate and return $\top$ if the state was initialized to be one, but it would diverge if the state was initialized to be two (i.e., $(\bot, \top)$) or three (i.e., $(\top, \top)$). Therefore, the answer to the reachability problem for this program would be yes in the first two cases and be no in the latter two cases.

Example 2. Next, let c_{ex2} be the following program, also adopted from [35].

$$\begin{array}{l}\texttt{let}\ r = (\texttt{with}\ h_{nd}\ \texttt{handle let}\ x = \texttt{if}\ \mathit{dec}\ ()\ \texttt{then}\ v_0\ \texttt{else}\ v_1\ \texttt{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{let}\ y = \texttt{if}\ \mathit{dec}\ ()\ \texttt{then}\ v_2\ \texttt{else}\ v_3\ \texttt{in}\ v_{xor}\ x\ y)\ \texttt{in} \\ \texttt{if}\ r\ \texttt{then}\ (\texttt{rec}\ f = \lambda\_.f\ ())\ ()\ \texttt{else return}\ \top\end{array}$$

where

$$\begin{array}{rl} v_{xor} & \triangleq \lambda x.\lambda y.\texttt{if}\ x\ \texttt{then}\ (\texttt{if}\ y\ \texttt{then}\ \bot\ \texttt{else}\ \top)\ \texttt{else}\ (\texttt{if}\ y\ \texttt{then}\ \top\ \texttt{else}\ \bot) \\ v_{or} & \triangleq \lambda x.\lambda y.\texttt{if}\ x\ \texttt{then}\ \top\ \texttt{else if}\ y\ \texttt{then}\ \top\ \texttt{else}\ \bot \\ h_{nd} & \triangleq \{\texttt{return}\ x = x, \mathit{dec}\ x\ k = v_{or}\ (k\ \top)\ (k\ \bot)\} \end{array}$$

Here, $v_0, v_1, v_2, v_3 \in \{\bot, \top\}$. The handler h_{nd} implements non-deterministic choice by the *dec* operation whose clause executes the given delimited continuation k, which is expected to take and return Booleans, with both $\top$ an $\bot$, and returns the disjunction of the two results. Therefore, the with-handle block will,

[2] Recall the syntactic sugar such as the notation $c_1; c_2$ remarked earlier.

for each of the four possibilities where x is bound to v_0 or v_1 and y is bound to v_2 or v_3, computes the exclusive-or of x and y, and takes the disjunction of the four results. Therefore, the with-handle block returns $\bot$ if the Booleans v_0, v_1, v_2, v_3 are all equal and otherwise returns $\top$, and the program c_{ex2} returns $\top$ in the former cases and diverges in the later cases (due to the infinite loop $(\texttt{rec}\ f = \lambda\_.f\ ())\ ()$). Thus, the answer to the reachability problem would be true in the former two cases (i.e., the four Booleans are all $\top$ or all $\bot$) and be no in the latter 14 cases.

$$b ::= \texttt{unit} \mid \texttt{bool}$$
$$\sigma ::= b \mid \sigma_1 \to \sigma_2$$

Fig. 3. The simple types.

Figure 3 defines the simple types. The base types are denoted by b. As usual, the function type constructor $\to$ associates to the right. Figure 4 shows the typing rules of the simple type system $\vdash_{\mathrm{st}}$. The type system is parameterized by a *signature* Σ that assigns types to the operation names. The rules (St-Unit) to (St-Let) are standard. The last four rules concern effect handlers, and they are also standard, matching those studied in prior work [35,20,17].[3] Importantly, (St-Hdlr) checks if a handler is well-typed by checking the well-typedness of the return clause as well as that of each operation clause. Note that the type of the return clause body, the types of the operation clause bodies, and the return types of the continuation parameters, are the same type σ'. This type σ' is called the *answer type*, and the fact that all the answer types in a handler are the same signifies that $\vdash_{\mathrm{st}}$ lacks *answer-type modification* (ATM). We shall show in Section 3 a type system that has ATM, $\vdash_{\mathrm{atm}}$, and show that the feature makes reachability decidable.

Example 3. Recall c_{ex1} and c_{ex2} from Examples 1 and 2. Both programs are $\vdash_{\mathrm{st}}$-typable. For c_{ex1}, the types of some subexpressions are as follows: $v_{tup} : \texttt{bool} \to \texttt{bool} \to \sigma_t$, $v_{fst}, v_{snd} : \sigma_t \to \texttt{bool}$, and $c_{inc} : \texttt{unit}$, where $\sigma_t = (\texttt{bool} \to \texttt{bool} \to \texttt{bool}) \to \texttt{bool}$. And, the typing for the handler can be given by $x : \sigma_t \vdash_{\mathrm{st}} \lambda s.x : \sigma_a$ for the return clause, and $x : \sigma_t, k : \texttt{unit} \to \sigma_a \vdash_{\mathrm{st}} \lambda s.k\ ()\ x : \sigma_a$ and $x : \texttt{unit}, k : \sigma_t \to \sigma_a \vdash_{\mathrm{st}} \lambda s.k\ s\ s : \sigma_a$ for the clauses of *set* and *get* respectively, where $\sigma_a = \sigma_t \to \texttt{unit}$. Note that the answer type is σ_a in all clauses, indicating that ATM is not needed to type the example. Similarly, c_{ex2} can be typed by typing the return clause as $x : \texttt{bool} \vdash_{\mathrm{st}} x : \texttt{bool}$, and the *dec* clause as $x : \texttt{unit}, k : \texttt{bool} \to \texttt{bool} \vdash_{\mathrm{st}} v_{or}\ (k\ \top)\ (k\ \bot) : \texttt{bool}$. Note that the answer type is `bool` in all clauses in this typing, again indicating that ATM is not needed for typing the example.

We say that a FPCF$_{\mathit{eff}}$ program c is $\vdash_{\mathrm{st}}$-*typable* if $\vdash_{\mathrm{st}} c : \sigma$ for some σ. We define (untyped) finitary PCF (without effect handlers) as the fragment of FPCF$_{\mathit{eff}}$ without `with` h `handle` c or $op\ v$, and we refer to the fragment

[3] Technically, [20,17] use weaker type systems that restrict (non-continuation) parameters of operations to base types. Our main result shows that ATM-typability makes reachability decidable even when no such restriction is imposed (cf. Sections 3.4 and 4 for further discussion).

$$\frac{}{\Gamma \vdash_{\mathrm{st}} () : \mathtt{unit}}\ (\textsc{St-Unit}) \quad \frac{v \in \{\top, \bot\}}{\Gamma \vdash_{\mathrm{st}} v : \mathtt{bool}}\ (\textsc{St-Bool}) \quad \frac{x : \sigma \in \Gamma}{\Gamma \vdash_{\mathrm{st}} x : \sigma}\ (\textsc{St-Var})$$

$$\frac{\Gamma, x : \sigma \vdash_{\mathrm{st}} c : \sigma'}{\Gamma \vdash_{\mathrm{st}} \lambda x.c : \sigma \to \sigma'}\ (\textsc{St-Lam}) \quad \frac{\Gamma, x : \sigma \vdash_{\mathrm{st}} v : \sigma}{\Gamma \vdash_{\mathrm{st}} \mathtt{rec}\ x = v : \sigma}\ (\textsc{St-Rec})$$

$$\frac{\Gamma \vdash_{\mathrm{st}} v : \mathtt{bool} \quad \Gamma \vdash_{\mathrm{st}} c_1 : \sigma \quad \Gamma \vdash_{\mathrm{st}} c_2 : \sigma}{\Gamma \vdash_{\mathrm{st}} \mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \sigma}\ (\textsc{St-If})$$

$$\frac{\Gamma \vdash_{\mathrm{st}} v_1 : \sigma \to \sigma' \quad \Gamma \vdash_{\mathrm{st}} v_2 : \sigma}{\Gamma \vdash_{\mathrm{st}} v_1\ v_2 : \sigma'}\ (\textsc{St-App}) \quad \frac{\Gamma \vdash_{\mathrm{st}} c_1 : \sigma \quad \Gamma, x : \sigma \vdash_{\mathrm{st}} c_2 : \sigma'}{\Gamma \vdash_{\mathrm{st}} \mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 : \sigma'}\ (\textsc{St-Let})$$

$$\frac{\Gamma \vdash_{\mathrm{st}} v : \sigma}{\Gamma \vdash_{\mathrm{st}} \mathtt{return}\ v : \sigma}\ (\textsc{St-Ret}) \quad \frac{\Sigma(op) = \sigma \to \sigma' \quad \Gamma \vdash_{\mathrm{st}} v : \sigma}{\Gamma \vdash_{\mathrm{st}} op\ v : \sigma'}\ (\textsc{St-Op})$$

$$\frac{\Gamma, x : \sigma \vdash_{\mathrm{st}} : c : \sigma' \quad \overline{\Gamma, x_i : \sigma_i, k_i : \sigma_i' \to \sigma' \vdash_{\mathrm{st}} c_i : \sigma'} \quad \overline{\Sigma(op_i) = \sigma_i \to \sigma_i'}}{\Gamma \vdash_{\mathrm{st}} \{\mathtt{return}\ x = c, \overline{op_i\ x_i\ k_i = c_i}\} : \sigma \to \sigma'}\ (\textsc{St-Hdlr})$$

$$\frac{\Gamma \vdash_{\mathrm{st}} h : \sigma \to \sigma' \quad \Gamma \vdash_{\mathrm{st}} c : \sigma}{\Gamma \vdash_{\mathrm{st}} \mathtt{with}\ h\ \mathtt{handle}\ c : \sigma'}\ (\textsc{St-Han})$$

Fig. 4. The typing rules of the simple type system $\vdash_{\mathrm{st}}$.

by FPCF. As mentioned in the introduction, the reachability problem for $\vdash_{\mathrm{st}}$-typable FPCF is decidable [27,16,41].

Theorem 1 ([27,16,41]). *Reachability is decidable for $\vdash_{\mathrm{st}}$-typable* FPCF.

By contrast, as also mentioned in the introduction, a recent paper [20] has shown that the reachability problem is undecidable for $\vdash_{\mathrm{st}}$-typable $\mathrm{FPCF}_{\mathit{eff}}$.[4]

Theorem 2 ([20,17]). *Reachability is undecidable for $\vdash_{\mathrm{st}}$-typable* $\mathrm{FPCF}_{\mathit{eff}}$.

3 Main Results

Figure 5 shows the ATM types. The base types, b, remain unchanged from those of ordinary simple types. A *value type*, denoted by τ, is either a base-type or a function type of the form $\tau' \to \rho$ where ρ is a *computation type*. A computation type is either of the form $\tau/\Box$ expressing a *pure* computation that returns a value of type τ, or of the form $\tau/\rho_1 \Rightarrow \rho_2$ expressing an *effectful* computation that changes the answer type from ρ_1 to ρ_2 and returns a value of type τ.

$$\tau ::= b \mid \tau \to \rho$$
$$\rho ::= \tau/\Box \mid \tau/\rho_1 \Rightarrow \rho_2$$

Fig. 5. The ATM types.

[4] An alternative proof is given in [17].

$$\frac{\Gamma \vdash_{\mathrm{atm}} c_1 : \tau_1/\square \quad \Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau_2/\square}{\Gamma \vdash_{\mathrm{atm}} \mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 : \tau_2/\square}\ (\text{T-LetP})$$

$$\frac{\Gamma \vdash_{\mathrm{atm}} c_1 : \tau_1/\rho_1 \Rightarrow \rho_1' \quad \Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau_2/\rho_2 \Rightarrow \rho_1}{\Gamma \vdash_{\mathrm{atm}} \mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 : \tau_2/\rho_2 \Rightarrow \rho_1'}\ (\text{T-LetIp})$$

$$\frac{\Gamma \vdash_{\mathrm{atm}} v : \tau}{\Gamma \vdash_{\mathrm{atm}} \mathtt{return}\ v : \tau/\square}\ (\text{T-Ret}) \quad \frac{\Sigma(op) = \tau \to \tau'/\rho_1 \Rightarrow \rho_2 \quad \Gamma \vdash_{\mathrm{atm}} v : \tau}{\Gamma \vdash_{\mathrm{atm}} op\ v : \tau'/\rho_1 \Rightarrow \rho_2}\ (\text{T-Op})$$

$$\frac{\overline{\Sigma(op_i) = \tau_i \to \tau_i'/\rho_i \Rightarrow \rho_i'} \quad \overline{\Gamma, x_i : \tau_i, k_i : \tau_i' \to \rho_i \vdash_{\mathrm{atm}} c_i : \rho_i'}}{\Gamma \vdash_{\mathrm{atm}} \{\mathtt{return}\ x = c, \overline{op_i\ x_i\ k_i = c_i}\}}\ (\text{T-Hdlr})$$

$$\frac{\Gamma \vdash_{\mathrm{atm}} h \quad \Gamma \vdash_{\mathrm{atm}} c : \tau/\rho \Rightarrow \rho' \quad \Gamma, x : \tau \vdash_{\mathrm{atm}} c' : \rho \quad \mathtt{return}\ x = c' \in h}{\Gamma \vdash_{\mathrm{atm}} \mathtt{with}\ h\ \mathtt{handle}\ c : \rho'}\ (\text{T-Han})$$

$$\frac{\Gamma \vdash_{\mathrm{atm}} v : \tau \quad \tau \leq \tau'}{\Gamma \vdash_{\mathrm{atm}} v : \tau'}\ (\text{T-VSub}) \quad \frac{\Gamma \vdash_{\mathrm{atm}} c : \rho \quad \rho \leq \rho'}{\Gamma \vdash_{\mathrm{atm}} c : \rho'}\ (\text{T-CSub})$$

Fig. 6. Representative typing rules of the ATM type system $\vdash_{\mathrm{atm}}$.

Figure 6 shows representative typing rules of the ATM type system $\vdash_{\mathrm{atm}}$. We refer to Appendix A for the complete set. The type system is essentially the ATM refinement type system proposed in [15], but without the refinement types aspect that is orthogonal to our paper. (T-LetP) stipulates that if both c_1 and c_2 are pure computations then the entire computation is also a pure one. By contrast, (T-LetIp) says that if c_1 is an effectful computation that changes the answer type from ρ_1 to ρ_1' and c_2 is an effectful computation that changes the answer type from ρ_2 to ρ_1, then the entire computation is an effectful computation that changes the answer type from ρ_2 to ρ_1'. Note that the answer types of the sub-computations are composed in a backward manner (cf. [15] for the explanation). (T-Ret) and (T-OP) are analogous to the corresponding rules (St-Ret) and (St-Ret) of $\vdash_{\mathrm{st}}$. In particular, the latter looks up the type of the operation in the signature Σ whose return computation type $\tau'/\rho_1 \Rightarrow \rho_2$ is an effectful one that changes the answer type from ρ_1 to ρ_2. (T-Hdlr) checks that a handler is well-typed. Note that, unlike (St-Hdlr) of $\vdash_{\mathrm{st}}$, the rule allows the operation clause bodies and their continuation parameters to have different answer types, signifying that $\vdash_{\mathrm{atm}}$ allows ATM. The return clause is typed in (T-Han) and it is also allowed to have a different answer type. Additionally, (T-Han) stipulates that the type of the with-handle block is changed from that of the return clause body, ρ, to ρ' by ATM. The last two rules, (T-VSub) and (T-CSub), are subsumption rules for the subtyping relation $\leq$.

A key subtyping rule is the following one that allows "embedding" a pure computation type into an effectful computation type:

$$\frac{\tau_1 \leq \tau_2 \quad \rho_1 \leq \rho_2}{\tau_1/\Box \leq \tau_2/\rho_1 \Rightarrow \rho_2} \text{ (S-EMBED)}$$

The rule signifies that a pure computation can always be used where an effectful one is expected. The remaining subtyping rules are defined inductively on the structure of the types. We refer to Appendix B for the complete set of subtyping rules. We say that a program c is $\vdash_{\text{atm}}$-*typable* if $\vdash_{\text{atm}} c : \tau/\Box$ for some τ.[5]

Example 4. Recall c_{ex1} and c_{ex2} from Examples 1 and 2. Recall that both programs are $\vdash_{\text{st}}$-typable as shown in Example 3. We show that both programs are also $\vdash_{\text{atm}}$-typable. For c_{ex1}, the typing for the handler can be given by $x : \tau_t \vdash_{\text{atm}} \lambda s.x : \rho_a$ for the return clause, $x : \sigma_t, k : \texttt{unit} \to \rho_a \vdash_{\text{atm}} \lambda s.k\ ()\ x : \rho_a$ for the *set* clause, and $x : \texttt{unit}, k : \tau_t \to \rho_a \vdash_{\text{atm}} \lambda x.k\ s\ s : \rho_a$ for the *get* clause, where $\tau_t = (\texttt{bool} \to (\texttt{bool} \to \texttt{bool}/\Box)/\Box) \to \texttt{bool}/\Box$ and $\rho_a = (\tau_t \to \texttt{unit}/\Box)/\Box$.

Typing these clauses does not need (T-LETIP) but only (T-LETP) because the computation in the clauses are all pure.[6] By contrast, the operation invocations of *get* and *set* will respectively be given effectful computation types $\tau_t/\rho_a \Rightarrow \rho_a$ and $\texttt{unit}/\rho_a \Rightarrow \rho_a$ by (T-OP). The body of the with-handle block will also be given an effectful computation type $\texttt{unit}/\rho_a \Rightarrow \rho_a$ by using (T-LETIP) to compose effectful computation types and giving the recursive function *loop* the type $\texttt{unit} \to \texttt{unit}/\rho_a \Rightarrow \rho_a$. Therefore, $\vdash_{\text{atm}} c_{ex1} : \texttt{bool}/\Box$. Similarly, c_{ex2} can be $\vdash_{\text{atm}}$-typed by typing the return clause and the *dec* clause as $x : \texttt{bool} \vdash_{\text{atm}} x : \rho_b$ and $x : \texttt{unit}, k : \texttt{bool} \to \rho_b \vdash_{\text{atm}} v_{or}\ (k\ \top)\ (k\ \bot) : \rho_b$, respectively, where $\rho_b = \texttt{bool}/\Box$. The invocations of *dec* in the with-handle-block body can be given the type $\texttt{bool}/\rho_b \Rightarrow \rho_b$ and so can the body itself.

Example 5. The previous example programs c_{ex1} and c_{ex2} were both $\vdash_{\text{st}}$-typable and $\vdash_{\text{atm}}$-typable. Here, we show an example of a program that *needs* ATM to be typed. That is, the program is $\vdash_{\text{atm}}$-typable but not $\vdash_{\text{st}}$-typable. Consider the following program c_{ex3}.

$$\texttt{let}\ r = (\texttt{with}\ \{\texttt{return}\ x = x, op\ x\ k = k\ x; \top\}\ \texttt{handle}\ op\ (); ())\ \texttt{in}$$
$$\texttt{if}\ r\ \texttt{then}\ \top\ \texttt{else}\ \bot$$

The program is $\vdash_{\text{atm}}$-typable by giving the invocation of *op* the type $\texttt{unit}/(\texttt{unit}/\Box \Rightarrow \texttt{bool}/\Box)$ and giving the variable r the type $\texttt{bool}$. However, it is not $\vdash_{\text{st}}$-typable because, in that type system, the return clause body needs to be given the type $\texttt{unit}$ whereas the *op* clause body needs to be given the type $\texttt{bool}$ and the rule for typing a handler (ST-HDLR) asserts that these types need to be the same.

[5] The restriction to pure types for $\vdash_{\text{atm}}$-typable programs is for simplicity. It lets us disregard the case the source program gets stuck with unhandled operation when showing the correctness of the CPS transformation. It can be shown that reachability remains decidable for $\vdash_{\text{atm}}$-typable programs even without the restriction.

[6] (T-LETP) is used when the trivial let bindings hidden by the notational convention are expanded (cf. Section 2).

As shown in the above example, there are programs typable with ATM but not without it. In fact, one may naturally expect allowing ATM makes more programs typable because it allows an operation invocation to change the type of a with-handle block from that of the return clause body to that of an operation clause body. Therefore, the following main result of the paper may come as a surprise because, as remarked before, reachability is undecidable for $\vdash_{\mathrm{st}}$-typable $\mathrm{FPCF}_{\mathit{eff}}$.

Theorem 3 (Decidability). *Reachability is decidable for* $\vdash_{\mathrm{atm}}$*-typable* $\mathrm{FPCF}_{\mathit{eff}}$.

The rest of this section is organized as follows. We prove Theorem 3 in Section 3.1 by presenting a novel typing-derivation-directed CPS transformation that transforms an $\vdash_{\mathrm{atm}}$-typable $\mathrm{FPCF}_{\mathit{eff}}$ program to a $\vdash_{\mathrm{st}}$-typable FPCF program. A corollary of Theorem 3 is the existence of a $\mathrm{FPCF}_{\mathit{eff}}$ program that is $\vdash_{\mathrm{st}}$-typable but not $\vdash_{\mathrm{atm}}$-typable, and we present a concrete example of such a program in Section 3.2 (Section 3.3 also has such an example). Section 3.3 describes another application of our CPS transformation. That is, we show there that every $\vdash_{\mathrm{atm}}$-typable recursive-definition-free $\mathrm{FPCF}_{\mathit{eff}}$ program terminates while the same does not hold for $\vdash_{\mathrm{st}}$-typable ones. Section 3.4 disproves a claim made in a recent paper [37] regarding what the paper calls *active effect handlers*.

3.1 Typing-Derivation-Directed CPS Transformation and the Proof of Theorem 3

We prove Theorem 3 by presenting a CPS transformation that transforms an $\vdash_{\mathrm{atm}}$-typable $\mathrm{FPCF}_{\mathit{eff}}$ program to a $\vdash_{\mathrm{st}}$-typable FPCF program. Then, Theorem 3 follows from the fact that reachability is decidable for $\vdash_{\mathrm{st}}$-typable FPCF programs.

We note that a CPS transformation for a language with effect handlers and ATM has already been proposed in a recent work by Kawamata et al. [15]. However, their CPS transformation requires higher-rank parametric polymorphism in an essential way to type the target of the transformation, and therefore, it is insufficient for our purpose because reachability for FPCF programs typable with a higher-rank parametric polymorphic type system is undecidable [40].

Our key observation is that the CPS transformation of [15] uses parametric polymorphism to allow a pure computation to be used in contexts where an effectful computation is expected. This is what subtyping is used for in $\vdash_{\mathrm{atm}}$. Based on the observation, and inspired by the CPS transformation proposed in a paper by Materzok and Biernacki [24], we propose a new CPS transformation for effect handlers that is subtyping-aware so that the target of the transformation can be typed without parametric polymorphism. The key idea, like that of [24], is to make the transformation be directed by the *typing derivation* of the source expression, instead of being only *type*-directed as more commonly seen for a CPS transformation.

For convenience, we extend $\vdash_{\mathrm{st}}$-typed FPCF with records. That is, we extend the syntax of expressions and simple types as follows.

$$v ::= \cdots \mid \{\overline{l_i = v_i}\} \qquad c ::= \cdots \mid v.l \qquad \sigma ::= \cdots \mid \{\overline{l_i : \sigma_i}\}$$

$$[\![b]\!] = b \quad [\![\tau \to \rho]\!] = [\![\tau]\!] \to [\![\rho]\!] \quad [\![\tau/\square]\!] = [\![\tau]\!]$$
$$[\![\tau/\rho_1 \Rightarrow \rho_2]\!] = [\![\Sigma]\!] \to ([\![\tau]\!] \to [\![\rho_1]\!]) \to [\![\rho_2]\!]$$
$$[\![\{\overline{op_i : \tau_i \to \tau_i'/\rho_i \Rightarrow \rho_i'}\}]\!] = \{\overline{op_i : [\![\tau_i]\!] \to ([\![\tau_i']\!] \to [\![\rho_i]\!]) \to [\![\rho_i']\!]}\}$$

Fig. 7. The CPS transformation of types.

The extension to the operational semantics and the typing rules is standard and is given in Appendix C. We note that the reachability for FPCF remains decidable with the record extension.[7]

Figure 7 shows the CPS transformation of types. As seen in the second to the last rule, an effectful computation is transformed to a function that takes a transformed signature (i.e., a record of the type $[\![\Sigma]\!]$), a transformed continuation of the type $[\![\tau]\!] \to [\![\rho_1]\!]$, and returns a value of the type $[\![\rho_2]\!]$. For a type environment Γ, we denote by $[\![\Gamma]\!]$ the CPS transformed environment $\{x : [\![\tau]\!] \mid x : \tau \in \Gamma\}$.

Next, we define the CPS transformation of subtyping derivations, which is of the form $[\![\tau_1 \leq \tau_2]\!]$ for subtyping of value types and $[\![\rho_1 \leq \rho_2]\!]$ for subtyping of computation types. An important case is the transformation of a derivation whose root is an instance of (S-Embed) shown below.

$$[\![\tau_1/\square \leq \tau_2/\rho_1 \Rightarrow \rho_2]\!] = \lambda x.\lambda h.\lambda k.[\![\rho_1 \leq \rho_2]\!]@(k@([\![\tau_1 \leq \tau_2]\!]@x)$$

where @ denotes a *static application* that is processed during the CPS transformation [13,15]. The complete set of the CPS transformation rules that concern subtyping derivations is given in Appendix D.

Finally, we define the CPS transformation of typing derivations, which is of the form $[\![\Gamma \vdash_{\mathrm{atm}} v : \tau]\!]$ for value expression typing and $[\![\Gamma \vdash_{\mathrm{atm}} c : \rho]\!]$ for computation expression typing. An interesting rule is one concerning the subsumption rule (T-CSub) shown below.

$$\left[\!\!\left[\frac{\Gamma \vdash_{\mathrm{atm}} c : \rho \quad \rho \leq \rho'}{\Gamma \vdash_{\mathrm{atm}} c : \rho'} \right]\!\!\right] = [\![\rho \leq \rho']\!] \mathbin{@} [\![\Gamma \vdash_{\mathrm{atm}} c : \rho]\!]$$

Note that it uses the transformation obtained from the subtyping $\rho \leq \rho'$ to properly CPS transform the source computation expression c that has been CPS transformed with respect to the sub-derivation $\Gamma \vdash_{\mathrm{atm}} c : \rho$. Another exemplifying transformation rule is one concerning the (T-Hdlr) rule for typing a handler shown below.

$$\left[\!\!\left[\frac{\overline{\Sigma(op_i) = \tau_i \to \tau_i'/\rho_i \Rightarrow \rho_i'} \quad \overline{\Gamma, x_i : \tau_i, k_i : \tau_i' \to \rho_i \vdash_{\mathrm{atm}} c_i : \rho_i'}}{\Gamma \vdash_{\mathrm{atm}} \{\mathtt{return}\ x = c, \overline{op_i\ x_i\ k_i = c_i}\}} \right]\!\!\right]$$
$$= \left\{\overline{op_i = \lambda x_i.\lambda k_i.[\![\Gamma, x_i : \tau_i, k_i : \tau_i' \to \rho_i \vdash_{\mathrm{atm}} c_i : \rho_i']\!]}\right\}$$

Note that the transformed result is a record mapping each operation name op_i to a function obtained by transforming the typing derivation for the operation

[7] This can be shown, for example, by encoding records as functions similarly to how it was done for tuples in Example 1.

clause body c_i. The rule is used in the CPS transformation corresponding to the typing rule (T-Han) for typing a with-handle block shown below.

$$\left[\!\!\left[\frac{\Gamma \vdash_{\mathrm{atm}} h \quad \Gamma \vdash_{\mathrm{atm}} c : \tau/\rho \Rightarrow \rho' \quad \Gamma, x : \tau \vdash_{\mathrm{atm}} c' : \rho \quad \mathtt{return}\ x = c' \in h}{\Gamma \vdash_{\mathrm{atm}} \mathtt{with}\ h\ \mathtt{handle}\ c : \rho'}\right]\!\!\right] \\ = [\![\Gamma \vdash_{\mathrm{atm}} c : \tau/\rho \Rightarrow \rho']\!] \mathbin{@} [\![\Gamma \vdash_{\mathrm{atm}} h]\!] \mathbin{@} \lambda x.\, [\![\Gamma, x : \tau \vdash_{\mathrm{atm}} c' : \rho]\!]$$

The rule uses the transformation rule corresponding to (T-Hdlr) mentioned above to transform the handler typing derivation $\Gamma \vdash_{\mathrm{atm}} h$ to a record of operations, transform the return clause typing derivation $\Gamma, x : \tau \vdash_{\mathrm{atm}} c' : \rho$, and apply the transformation of the typing derivation for the with-handle block body $\Gamma \vdash_{\mathrm{atm}} c : \tau/\rho \Rightarrow \rho'$, which would be a function of the type $[\![\Sigma]\!] \to ([\![\tau]\!] \to [\![\rho]\!]) \to [\![\rho']\!]$, to the former and the λ abstraction of the latter. The passed record will be looked up in the transformation of an operation invocation, as seen in the following transformation rule corresponding to (T-Op).

$$\left[\!\!\left[\frac{\Sigma(op) = \tau \to \tau'/\rho_1 \Rightarrow \rho_2 \quad \Gamma \vdash_{\mathrm{atm}} v : \tau}{\Gamma \vdash_{\mathrm{atm}} op\ v : \tau'/\rho_1 \Rightarrow \rho_2}\right]\!\!\right] = \lambda h.\lambda k.h.op\ [\![\Gamma \vdash_{\mathrm{atm}} v : \tau]\!]\ k$$

We refer to Appendix E for the complete set of transformation rules that concern typing derivations.

We show the correctness of our CPS transformation. First, we show that the transformed program is a $\vdash_{\mathrm{st}}$-typable FPCF program, which follows immediately from the following theorem that can be proven by induction on (sub)typing derivations, and the fact that the right hands of the transformation rules do not contain effect handlers.

Theorem 4 (Typability Preservation). *The following holds.*

1. *If* $\tau \leq \tau'$ *then* $\vdash_{\mathrm{st}} [\![\tau \leq \tau']\!] : [\![\tau]\!] \to [\![\tau']\!]$.
2. *If* $\rho \leq \rho'$ *then* $\vdash_{\mathrm{st}} [\![\rho \leq \rho']\!] : [\![\rho]\!] \to [\![\rho']\!]$.
3. *If* $\Gamma \vdash_{\mathrm{atm}} v : \tau$ *then* $[\![\Gamma]\!] \vdash_{\mathrm{st}} [\![\Gamma \vdash_{\mathrm{atm}} v : \tau]\!] : [\![\tau]\!]$.
4. *If* $\Gamma \vdash_{\mathrm{atm}} c : \rho$ *then* $[\![\Gamma]\!] \vdash_{\mathrm{st}} [\![\Gamma \vdash_{\mathrm{atm}} c : \rho]\!] : [\![\rho]\!]$.

Next, we show that the transformation preserves reachability. That is, if c is $\vdash_{\mathrm{atm}}$-typable by a derivation $\vdash_{\mathrm{atm}} c : \tau/\Box$, then the answer to the reachability problem for c is the same as that for $[\![\vdash_{\mathrm{atm}} c : \tau/\Box]\!]$. This is shown by the following simulation theorem. Let $\to^+$ denote the transitive closure of $\to$.

Theorem 5 (Simulation). *Suppose* $\vdash_{\mathrm{atm}} c : \tau/\Box$. *The following holds.*

1. *If* $c \to^* \mathtt{return}\ v$ *then* $\vdash_{\mathrm{atm}} v : \tau$ *and* $[\![\vdash_{\mathrm{atm}} c : \tau/\Box]\!] \to^+ [\![\vdash_{\mathrm{atm}} v : \tau]\!]$.
2. *If* $[\![\vdash_{\mathrm{atm}} c : \tau/\Box]\!] \to^+ \mathtt{return}\ v'$ *then there is* v *such that* $\vdash_{\mathrm{atm}} v : \tau$, $[\![\vdash_{\mathrm{atm}} v : \tau]\!] = v'$, *and* $c \to^* \mathtt{return}\ v$.

We refer to Appendix F for the proof. Therefore, given an $\vdash_{\mathrm{atm}}$-typable c, we can decide the reachability of c by deciding the reachability of $[\![\vdash_{\mathrm{atm}} c : \tau/\Box]\!]$, because $[\![\vdash_{\mathrm{atm}} c : \tau/\Box]\!]$ is a $\vdash_{\mathrm{st}}$-typable FPCF program by Theorem 4 and the reachability problem of $\vdash_{\mathrm{st}}$-typable FPCF is decidable. Thus, reachability for $\vdash_{\mathrm{atm}}$-typable FPCF$_{\mathit{eff}}$ is decidable. This completes the proof of Theorem 3.

3.2 A Concrete Example of a $\vdash_{\mathbf{st}}$-Typable but $\vdash_{\mathbf{atm}}$-Untypable Program

A corollary of our decidability result (Theorem 3) is that there are $\vdash_{\mathrm{st}}$-typable programs that are not $\vdash_{\mathrm{atm}}$-typable, which may seem counter-intuitive because one may naturally think that allowing ATM can only make more programs typable. We give a concrete example of such a program. Let

$$c_{ex4} \triangleq \texttt{rec}\ f = \lambda z.\texttt{with}\ h\ \texttt{handle}\ f\ ()\quad h \triangleq \{\texttt{return}\ x = op\ (), op\ x\ k = ()\}$$

This program c_{ex4} is $\vdash_{\mathrm{st}}$-typable. Namely, $\vdash_{\mathrm{st}} c_{ex4} : \texttt{unit} \to \texttt{unit}$ with $\Sigma(op) = \texttt{unit} \to \texttt{unit}$. The recursively defined function f would also be given the type $\texttt{unit} \to \texttt{unit}$. We now show that c_{ex4} is $\vdash_{\mathrm{atm}}$-untypable. Suppose for contradiction that it is $\vdash_{\mathrm{atm}}$-typable. It must be the case that the recursive function f is given the type $\texttt{unit} \to \rho_f$ for some ρ_f. Because the body of the with-handle block is $f\ ()$, by (T-Han), this ρ_f must satisfy $\rho_f \leq \tau/\rho_1 \Rightarrow \rho_2$ for some τ, ρ_1, and ρ_2. However, by (T-Han) again, ρ_2 must be a subtype of the with-handle block and thus a subtype of ρ_f by (T-Rec). That is, we have the subtyping relation $\rho_2 \leq \tau/\rho_1 \Rightarrow \rho_2$. We state the following general property of ATM subtyping, which will be used in our argument.

Lemma 1. *If $\rho \leq \tau/\rho' \Rightarrow \rho$ then ρ' is pure.*

Proof. By induction on the structure of ρ. □

Therefore, ρ_1 must be pure. This is not possible because ρ_1 must be a supertype of the type of the return clause, but the return clause invokes the operation op and therefore must be given an effectful type. Thus, c_{ex4} is not $\vdash_{\mathrm{atm}}$-typable.

3.3 Termination and Non-Termination for $\vdash_{\mathbf{atm}}$-Typable and $\vdash_{\mathbf{st}}$-Typable Programs

The example of a $\vdash_{\mathrm{st}}$-typable but $\vdash_{\mathrm{atm}}$-untypable program given in Section 3.2 used a recursive definition in an essential way. A natural question is whether this is always the case, that is, recursive definitions are necessary for a program to be $\vdash_{\mathrm{st}}$-typable but not $\vdash_{\mathrm{atm}}$-typable. We answer the question negatively by presenting a recursive-definition-free program that is $\vdash_{\mathrm{st}}$-typable but not $\vdash_{\mathrm{atm}}$-typable. We do this by presenting a result that may be of independent interest and says that any $\vdash_{\mathrm{atm}}$-typable recursive-definition-free program terminates whereas the same is not true for the $\vdash_{\mathrm{st}}$-typable ones.

Consider the following recursive-definition-free program c_{ex5}:

$$(\texttt{with}\ h_{st}\ \texttt{handle}\ \texttt{let}\ f = \lambda x.get\ ()\ ()\ \texttt{in}\ (set\ f; f\ ()))\ \lambda x.()$$

where h_{st} is the mutable state handler from Example 1. This program essentially implements the textbook encoding of an infinite loop by a mutable state (i.e.,

Landin's knot), and is non-terminating. Indeed, we have

$$\begin{aligned} c_{ex5} &\to (\lambda s.(\lambda\_.\texttt{with}\ h_{st}\ \texttt{handle}\ v_f\ ())\ ()\ v_f)\ \lambda x.() \\ &\to (\lambda\_.\texttt{with}\ h_{st}\ \texttt{handle}\ v_f\ ())\ ()\ v_f \to (\texttt{with}\ h_{st}\ \texttt{handle}\ v_f\ ())\ v_f \\ &\to (\texttt{with}\ h_{st}\ \texttt{handle}\ \mathit{get}\ ()\ ())\ v_f \to (\lambda s.(\lambda x.\texttt{with}\ h_{st}\ \texttt{handle}\ x\ ())\ s\ s)\ v_f \\ &\to (\lambda x.\texttt{with}\ h_{st}\ \texttt{handle}\ x\ ())\ v_f\ v_f \to (\texttt{with}\ h_{st}\ \texttt{handle}\ v_f\ ())\ v_f \end{aligned}$$

where $v_f = \lambda\_.\mathit{get}\,()\,()$. Because the last term is the same as the fourth term, the evaluation diverges. The program c_{ex5} is $\vdash_{\mathrm{st}}$-typable. Namely, it can be $\vdash_{\mathrm{st}}$-typed by giving *get* the type $\texttt{unit} \to \sigma_t$ and *set* the type $\sigma_t \to \texttt{unit}$ in the operation signature, where $\sigma_t = \texttt{unit} \to \texttt{unit}$. The typing of the return, *get*, and *set* clauses can be done in the same way as in Example 3 except for using the above σ_t for the σ_t there. The with-handle block can be given the type `unit` with these types of *get* and *set*. However, c_{ex5} is not $\vdash_{\mathrm{atm}}$-typable. This follows from the theorem below which can be easily shown by using our CPS transformation.

Theorem 6. *Every* $\vdash_{\mathrm{atm}}$*-typable* FPCF$_{\mathit{eff}}$ *program without recursive definitions terminates.*

Proof. Let c be an $\vdash_{\mathrm{atm}}$-typable FPCF$_{\mathit{eff}}$ program without recursive definitions, and let $c' = [\![\vdash_{\mathrm{atm}} c : \tau/\Box]\!]$. Because our CPS transformation does not add new recursive definitions, c' also does not contain recursive definitions (in addition to not containing effect handlers). By the results shown in Section 3.1, c' is $\vdash_{\mathrm{st}}$-typable and c' terminates iff c terminates. Because simply-typed λ calculus (without recursive definitions or effect handlers) is terminating, c' must terminate and therefore so must c. □

3.4 Number of Active Effect Handlers Does Not Capture Decidability

A recent paper [37] introduced a notion called *active effect handlers*. Their paper claims that the boundedness of the number of active effect handlers characterizes the decidability of reachability, and that the reason why ATM ensures the decidability is because it ensures that this number is bounded. In this section, we disprove this claim by presenting a class of programs with only a bounded number of active effect handler) but whose reachability is nonetheless undecidable.

Let *succ* be an operation, x_0, x_1, and $f_0, \ldots, f_n$ be variables where $n \geq 0$. Let us define the set of computation expressions $\mathit{Inst}^n \triangleq \{c_{inc}^{i,j} \mid i \in \{0,1\}, j \in \{0,\ldots,n\}\} \cup \{c_{dec}^{i,j,m} \mid i \in \{0,1\}, j,m \in \{0,\ldots,n\} \cup \{()\}$ where

$$\begin{aligned} c_{inc}^{i,j} &\triangleq \texttt{let}\ x_i = \lambda y.\mathit{succ}\ x_i\ \texttt{in}\ f_j\ x_0\ x_1 \\ c_{dec}^{i,j,m} &\triangleq \texttt{with}\ \{\texttt{return}\ x = f_j\ x_0\ x_1, \mathit{succ}\ x_i\ k = f_m\ x_0\ x_1\}\ \texttt{handle}\ x_i\ () \end{aligned}$$

Then, let MM^n be the class of programs of the form

$$\texttt{mrec}\ f_0 = \lambda x_0.\lambda x_1.c_0\ \texttt{and}\ \ldots\ \texttt{and}\ f_n = \lambda x_0.\lambda x_1.c_n\ \texttt{in}\ (f_0\ (\lambda x.())\ \lambda x.()); \top$$

where each $c_i \in \mathit{Inst}^n$, and the mutual recursion $\texttt{mrec}\ f_0 = v_0\ \texttt{and}\ \ldots\ \texttt{and}\ f_n = v_n\ \texttt{in}\ c$ is syntactic sugar defined inductively by:

$$\texttt{let}\ f_0 = \texttt{rec}\ f_0 = (\texttt{mrec}\ f_1 = v_1\ \texttt{and}\ \ldots\ \texttt{and}\ f_n = v_n\ \texttt{in}\ v_0)\ \texttt{in}\ \ldots$$
$$\texttt{let}\ f_n = \texttt{rec}\ f_n = (\texttt{mrec}\ f_0 = v_0\ \texttt{and}\ \ldots\ \texttt{and}\ f_{n-1} = v_{n-1}\ \texttt{in}\ v_n)\ \texttt{in}\ c$$

Let $\mathit{MM} = \bigcup_{n\in\mathbb{N}} \mathit{MM}^n$. We refer to Appendix G for the proof of the following theorem.

Theorem 7. *Reachability for MM is undecidable.*

We note that *MM* is $\vdash_{\mathrm{st}}$-typable. Indeed, any program in *MM* can be $\vdash_{\mathrm{st}}$-typed by giving the operation *succ* the type $\sigma_{nat} \to \texttt{unit}$ and each recursive function f_i the type $\sigma_{nat} \to \sigma_{nat} \to \texttt{unit}$ where $\sigma_{nat} = \texttt{unit} \to \texttt{unit}$.

Next, we recall the notion of *active effect handlers* from [37]. Concretely, the number of active effect handlers is said to be *bounded* for a program if there is a non-negative integer n such that the evaluation of the program only yields intermediate expressions of the form

$$\ldots(\texttt{with}\ h_1\ \texttt{handle}\ldots(\texttt{with}\ h_m\ \texttt{handle}\ c)\ \ldots)\ldots$$

for $m \leq n$ where c does not contain a with-handle block. Roughly, the number of active effect handlers measures the number of pending effect handlers on the call stack (when effect handlers are implemented by using a call stack, as often done in real implementations).

It is easy to see that the number of active effect handlers for any program in *MM* is bounded (in fact, by one), because a with-handle block will only appear in the evaluation by a calling a recursive function whose body is some c_{dec}, but then a subsequent recursive function call can happen only by replacing this with-handle block by the body of the return clause or the *succ* clause. Because reachability for *MM* is undecidable as shown in Theorem 7, this disproves the claim that the boundedness of the number of active effect handlers characterizes the decidability of reachability for finitary PCF with effect handlers.

Additionally, a recent paper [20] contains a result stating that the reachability problem becomes decidable when the operation clauses are restricted to only use the given delimited continuation at tail positions. This may appear to contradict Theorem 7 because the operation (i.e., *succ*) clause in *MM* does not use the given delimited continuation. But it actually does not, because [20] restricts the (non-continuation) parameters of operations to base types and thus disallows programs like *MM*.[8] The main result of our paper (Theorem 3) shows that such restrictions on operation parameter types or delimited continuation usage are not need for ATM to ensure the decidability of reachability.

4 Related Work

As mentioned in the introduction, our work is inspired by the prior research on reachability for finitary PCF (without effect handlers). The problem was shown

[8] The same restriction is used in [17].

to be decidable [27,16,41], and the result has served as a foundation of methods for verifying infinite-state higher-order-recursive programs by incorporating techniques like predicate abstraction and CEGAR to abstract infinite data to finite domains [4,7,3,39,28,18,42,36]. It is worth noting that such studies have extended beyond just reachability (i.e., safety properties) and have also lead to methods for verifying liveness properties by incorporating techniques like binary reachability analysis and automata-theoretic verification method [43,34,9,8,19,26].

Inspired by this success and the popularity effect handlers, a recent paper by Dal Lago and Ghyselen [20] has investigated the decidability of reachability for finitary PCF extended with effect handlers, and has found that the problem is undecidable. An alternative proof of this undecidability is also given in a recent paper by Kobayashi [17]. In this paper, we have shown that this undecidability comes from the way the standard type systems for effect handlers are designed, and that, surprisingly, the problem becomes decidable when the type system is extended to allow ATM. As remarked in Section 3.4, [20] contains a result stating that the reachability problem becomes decidable if the operation clauses are restricted to only use the captured delimited continuation at tail positions. However, as we have shown there, this decidability result crucially relies on the fact that operation parameters are restricted to base types in [20] because allowing arbitrary (simple) types for operation parameters makes reachability undecidable even with the restriction on the delimited continuation usage. The decidability result of our paper shows that such restrictions on operation parameter types or delimited continuation usage are not needed for ATM to ensure the decidability of reachability.

Our decidability result was proven by a novel typing-derivation-directed CPS transformation that transforms an ATM-typable program with effect handlers to a simply-typable program without effect handlers. Our CPS transformation is based on the one proposed by Kawamata et al. [15] but using typing-derivation dependency to eliminate parametric polymorphism. The elimination was crucial for reducing the problem to the decidable problem of reachability for (non-polymorphic) finitary PCF. As mentioned before, [15] used higher-rank parametric polymorphism in an essential way to allow pure computations to be used in contexts where effectful ones are expected, and we have observed that this is precisely what subtyping is used for by the ATM type system. We have adopted the ideas from the (sub)typing-derivation-directed CPS transformation for delimited control presented in the paper by Materzok and Biernacki [24] to design a new (sub)typing-derivation-directed CPS transformation for effect handlers. The CPS transformation of [24] does not consider effect handlers, and, as mentioned above, the CPS transformation of [15] requires parametric polymorphism. Our new CPS transformation makes a novel combination of the ideas from these prior CPS transformations.

A recent paper by Sekiyama and Unno [37] proves a similar but strictly weaker decidability result which essentially says that the typability in an ATM type system lacking subtyping is sufficient for decidability of the reachability for finitary PCF with effect handlers. Our paper proves a stronger result that

says that typability in the ATM type system that supports subtyping is sufficient for the decidability, and their result follows as an easy corollary of our result. We note that subtyping makes a significant difference for an ATM type system. Indeed, without subtyping, an ATM type system cannot, for example, type a program that uses a pure function in contexts expecting effectful computations with different answer-type modifications. The lack of subtyping allowed their paper to use a more straightforward CPS transformation obtained by simply dropping parametric polymorphism from the CPS transformation of [15]. By contrast, our CPS transformation makes a novel use of the ideas from the (sub)typing-derivation-driven CPS transformation of [24] to eliminate parametric polymorphism without losing the support for subtyping. Additionally, we have disproved an erroneous claim made in [37] regarding active effect handlers. Namely, we have shown that the notion does not capture the decidability of reachability for programs with effect handlers, and that boundedness of their number is not the reason why ATM ensures the decidability of reachability.

5 Conclusion

We have studied the reachability problem for finitary PCF extended with effect handlers. Recent work [20,17] have shown that the problem is undecidable, in stark contrast to the case without effect handlers for which the problem is known to be decidable [27,16,41]. In this paper, we have shown that the undecidability comes from the way the standard type systems for effect handlers are designed. Concretely, we have shown that, perhaps surprisingly, extending the type system to allow ATM recovers decidability. A corollary of our decidability result is that, perhaps counter-intuitively, there are program that are typable without ATM but untypable with it, and we have shown concrete examples of such programs. Our decidability result was proven by a novel typing-derivation-driven CPS transformation, and as another application of the CPS transformation, we have shown that every ATM-typable recursive-definition-free finitary PCF programs terminates while the same does not hold for the simply-typable ones. Finally, we have disproved a claim made in a recent paper [37] by showing that active effect handlers do not characterize the decidability of reachability for finitary PCF with effect handlers. We foresee our decidability result to lay a foundation for developing verification methods for programs with effect handlers, just as the decidability result for reachability of finitary PCF has done such for programs without effect handlers.

Acknowledgments. We thank the anonymous reviewers for their suggestions. We thank Yukiyoshi Kameyama, Naoki Kobayashi, Taro Sekiyama, and Hiroshi Unno for discussions on the work. This work was supported by JSPS KAKENHI Grant Numbers JP23K24826 and JP20K20625.

References

1. Alvarez-Picallo, M., Freund, T., Ghica, D.R., Lindley, S.: Effect handlers for C via coroutines. Proc. ACM Program. Lang. **8**(OOPSLA2), 2462–2489 (2024), `https:`

//doi.org/10.1145/3689798
2. Asai, K.: On typing delimited continuations: three new solutions to the printf problem. High. Order Symb. Comput. **22**(3), 275–291 (2009), https://doi.org/10.1007/s10990-009-9049-5
3. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001. pp. 203–213. ACM (2001), https://doi.org/10.1145/378795.378846
4. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002. pp. 1–3. ACM (2002), https://doi.org/10.1145/503272.503274
5. Bauer, A.: What is algebraic about algebraic effects and handlers? CoRR **abs/1807.05923** (2018), http://arxiv.org/abs/1807.05923
6. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effect handlers for the masses. Proc. ACM Program. Lang. **2**(OOPSLA), 111:1–111:27 (2018), https://doi.org/10.1145/3276481
7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1855, pp. 154–169. Springer (2000), https://doi.org/10.1007/10722167_15
8. Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., Vardi, M.Y.: Proving that programs eventually do something good. In: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007. pp. 265–276. ACM (2007), https://doi.org/10.1145/1190216.1190257
9. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006. pp. 415–426. ACM (2006), https://doi.org/10.1145/1133981.1134029
10. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990. pp. 151–160. ACM (1990), https://doi.org/10.1145/91556.91622
11. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1855, pp. 232–247. Springer (2000), https://doi.org/10.1007/10722167_20
12. Ghica, D.R., Lindley, S., Bravo, M.M., Piróg, M.: High-level effect handlers in C++. Proc. ACM Program. Lang. **6**(OOPSLA2), 1639–1667 (2022), https://doi.org/10.1145/3563445
13. Hillerström, D., Lindley, S., Atkey, R., Sivaramakrishnan, K.C.: Continuation passing style for effect handlers. In: 2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK. LIPIcs, vol. 84, pp. 18:1–18:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017), https://doi.org/10.4230/LIPIcs.FSCD.2017.18
14. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA

- September 25 - 27, 2013. pp. 145–158. ACM (2013), https://doi.org/10.1145/2500365.2500590
15. Kawamata, F., Unno, H., Sekiyama, T., Terauchi, T.: Answer refinement modification: Refinement type system for algebraic effects and handlers. Proc. ACM Program. Lang. **8**(POPL), 115–147 (2024), https://doi.org/10.1145/3633280
16. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009. pp. 416–428. ACM (2009), https://doi.org/10.1145/1480881.1480933
17. Kobayashi, N.: On decidable and undecidable extensions of simply typed lambda calculus. Proc. ACM Program. Lang. **9**(POPL), 1136–1166 (2025), https://doi.org/10.1145/3704875
18. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 222–233. ACM (2011), https://doi.org/10.1145/1993498.1993525
19. Kuwahara, T., Terauchi, T., Unno, H., Kobayashi, N.: Automatic termination verification for higher-order functional programs. In: Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8410, pp. 392–411. Springer (2014), https://doi.org/10.1007/978-3-642-54833-8_21
20. Lago, U.D., Ghyselen, A.: On model-checking higher-order effectful programs. Proc. ACM Program. Lang. **8**(POPL), 2610–2638 (2024), https://doi.org/10.1145/3632929
21. Leijen, D.: Implementing algebraic effects in C - "monads for free in C". In: Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10695, pp. 339–363. Springer (2017), https://doi.org/10.1007/978-3-319-71237-6_17
22. Levy, P.B.: Call-By-Push-Value: A Functional/Imperative Synthesis, Semantics Structures in Computation, vol. 2. Springer (2004)
23. Loader, R.: Finitary PCF is not decidable. Theor. Comput. Sci. **266**(1-2), 341–364 (2001), https://doi.org/10.1016/S0304-3975(00)00194-8
24. Materzok, M., Biernacki, D.: Subtyping delimited continuations. In: Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011. pp. 81–93. ACM (2011), https://doi.org/10.1145/2034773.2034786
25. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs, NJ (1967)
26. Murase, A., Terauchi, T., Kobayashi, N., Sato, R., Unno, H.: Temporal verification of higher-order functional programs. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 57–68. ACM (2016), https://doi.org/10.1145/2837614.2837667
27. Ong, C.L.: On model-checking trees generated by higher-order recursion schemes. In: 21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 Au-

gust 2006, Seattle, WA, USA, Proceedings. pp. 81–90. IEEE Computer Society (2006), https://doi.org/10.1109/LICS.2006.38
28. Ong, C.L., Ramsay, S.J.: Verifying higher-order functional programs with pattern-matching algebraic data types. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 587–598. ACM (2011), https://doi.org/10.1145/1926385.1926453
29. Plotkin, G.D., Power, J.: Adequacy for algebraic effects. In: Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2030, pp. 1–24. Springer (2001), https://doi.org/10.1007/3-540-45315-6_1
30. Plotkin, G.D., Power, J.: Notions of computation determine monads. In: Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2303, pp. 342–356. Springer (2002), https://doi.org/10.1007/3-540-45931-6_24
31. Plotkin, G.D., Power, J.: Algebraic operations and generic effects. Appl. Categorical Struct. **11**(1), 69–94 (2003), https://doi.org/10.1023/A:1023064908962
32. Plotkin, G.D., Pretnar, M.: A logic for algebraic effects. In: Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA. pp. 118–129. IEEE Computer Society (2008), https://doi.org/10.1109/LICS.2008.45
33. Plotkin, G.D., Pretnar, M.: Handlers of algebraic effects. In: Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5502, pp. 80–94. Springer (2009), https://doi.org/10.1007/978-3-642-00590-9_7
34. Podelski, A., Rybalchenko, A.: Transition invariants. In: 19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings. pp. 32–41. IEEE Computer Society (2004), https://doi.org/10.1109/LICS.2004.1319598
35. Pretnar, M.: An introduction to algebraic effects and handlers. invited tutorial paper. In: The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015. Electronic Notes in Theoretical Computer Science, vol. 319, pp. 19–35. Elsevier (2015), https://doi.org/10.1016/j.entcs.2015.12.003
36. Ramsay, S.J., Neatherway, R.P., Ong, C.L.: A type-directed abstraction refinement approach to higher-order model checking. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 61–72. ACM (2014), https://doi.org/10.1145/2535838.2535873
37. Sekiyama, T., Unno, H.: Higher-order model checking of effect-handling programs with answer-type modification. Proc. ACM Program. Lang. **8**(OOPSLA2), 2662–2691 (2024), https://doi.org/10.1145/3689805
38. Sivaramakrishnan, K.C., Dolan, S., White, L., Kelly, T., Jaffer, S., Madhavapeddy, A.: Retrofitting effect handlers onto OCaml. In: PLDI '21: 42nd ACM SIGPLAN

International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 206–221. ACM (2021), https://doi.org/10.1145/3453483.3454039
39. Terauchi, T.: Dependent types from counterexamples. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010. pp. 119–130. ACM (2010), https://doi.org/10.1145/1706299.1706315
40. Tsukada, T., Kobayashi, N.: Untyped recursion schemes and infinite intersection types. In: Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6014, pp. 343–357. Springer (2010), https://doi.org/10.1007/978-3-642-12032-9_24
41. Tsukada, T., Kobayashi, N.: Complexity of model-checking call-by-value programs. In: Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8412, pp. 180–194. Springer (2014), https://doi.org/10.1007/978-3-642-54830-7_12
42. Unno, H., Terauchi, T., Kobayashi, N.: Automating relatively complete verification of higher-order functional programs. In: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 75–86. ACM (2013), https://doi.org/10.1145/2429069.2429081
43. Vardi, M.Y.: Verification of concurrent programs: The automata-theoretic framework. Ann. Pure Appl. Log. **51**(1-2), 79–98 (1991), https://doi.org/10.1016/0168-0072(91)90066-U
44. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. **115**(1), 38–94 (1994), https://doi.org/10.1006/inco.1994.1093
45. Xie, N., Leijen, D.: Effect handlers in Haskell, evidently. In: Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2020, Virtual Event, USA, August 7, 2020. pp. 95–108. ACM (2020), https://doi.org/10.1145/3406088.3409022

A $\vdash_{\mathbf{atm}}$ Typing Rules

Figure 8 shows the complete set of $\vdash_{\mathrm{atm}}$ typing rules.

B $\vdash_{\mathbf{atm}}$ Subtyping Rules

Figure 9 shows the complete set of $\vdash_{\mathrm{atm}}$ subtyping rules.

C Extending $\vdash_{\mathbf{st}}$-Typable FPCF with Records

We extend the operational semantics by adding the following rule.

$$\frac{}{\{\overline{l_i = v_i}\}.l_i \to v_i}\ (\text{E-Proj})$$

$$\frac{}{\Gamma \vdash_{\mathrm{atm}} () : \mathtt{unit}}\ (\text{T-Unit}) \quad \frac{v \in \{\top, \bot\}}{\Gamma \vdash_{\mathrm{atm}} v : \mathtt{bool}}\ (\text{T-Bool})$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash_{\mathrm{atm}} x : \tau}\ (\text{T-Var}) \quad \frac{\Gamma, x : \tau \vdash_{\mathrm{atm}} c : \rho}{\Gamma \vdash_{\mathrm{atm}} \lambda x.c : \tau \to \rho}\ (\text{T-Lam}) \quad \frac{\Gamma, x : \tau \vdash_{\mathrm{atm}} v : \tau}{\Gamma \vdash_{\mathrm{atm}} \mathtt{rec}\ x = v : \tau}\ (\text{T-Rec})$$

$$\frac{\Gamma \vdash_{\mathrm{atm}} v : \mathtt{bool} \quad \Gamma \vdash_{\mathrm{atm}} c_1 : \rho \quad \Gamma \vdash_{\mathrm{atm}} c_2 : \rho}{\Gamma \vdash_{\mathrm{atm}} \mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \rho}\ (\text{T-If})$$

$$\frac{\Gamma \vdash_{\mathrm{atm}} v_1 : \tau \to \rho \quad \Gamma \vdash_{\mathrm{atm}} v_2 : \tau}{\Gamma \vdash_{\mathrm{atm}} v_1\ v_2 : \rho}\ (\text{T-App})$$

$$\frac{\Gamma \vdash_{\mathrm{atm}} c_1 : \tau_1/\Box \quad \Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau_2/\Box}{\Gamma \vdash_{\mathrm{atm}} \mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 : \tau_2/\Box}\ (\text{T-LetP})$$

$$\frac{\Gamma \vdash_{\mathrm{atm}} c_1 : \tau_1/\rho_1 \Rightarrow \rho_1' \quad \Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau_2/\rho_2 \Rightarrow \rho_1}{\Gamma \vdash_{\mathrm{atm}} \mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 : \tau_2/\rho_2 \Rightarrow \rho_1'}\ (\text{T-LetIp})$$

$$\frac{\Gamma \vdash_{\mathrm{atm}} v : \tau}{\Gamma \vdash_{\mathrm{atm}} \mathtt{return}\ v : \tau/\Box}\ (\text{T-Ret}) \quad \frac{\Sigma(op) = \tau \to \tau'/\rho_1 \Rightarrow \rho_2 \quad \Gamma \vdash_{\mathrm{atm}} v : \tau}{\Gamma \vdash_{\mathrm{atm}} op\ v : \tau'/\rho_1 \Rightarrow \rho_2}\ (\text{T-Op})$$

$$\frac{\overline{\Sigma(op_i) = \tau_i \to \tau_i'/\rho_i \Rightarrow \rho_i'} \quad \overline{\Gamma, x_i : \tau_i, k_i : \tau_i' \to \rho_i \vdash_{\mathrm{atm}} c_i : \rho_i'}}{\Gamma \vdash_{\mathrm{atm}} \{\mathtt{return}\ x = c, \overline{op_i\ x_i\ k_i = c_i}\}}\ (\text{T-Hdlr})$$

$$\frac{\Gamma \vdash_{\mathrm{atm}} h \quad \Gamma \vdash_{\mathrm{atm}} c : \tau/\rho \Rightarrow \rho' \quad \Gamma, x : \tau \vdash_{\mathrm{atm}} c' : \rho \quad \mathtt{return}\ x = c' \in h}{\Gamma \vdash_{\mathrm{atm}} \mathtt{with}\ h\ \mathtt{handle}\ c : \rho'}\ (\text{T-Han})$$

$$\frac{\Gamma \vdash_{\mathrm{atm}} v : \tau \quad \tau \leq \tau'}{\Gamma \vdash_{\mathrm{atm}} v : \tau'}\ (\text{T-VSub}) \quad \frac{\Gamma \vdash_{\mathrm{atm}} c : \rho \quad \rho \leq \rho'}{\Gamma \vdash_{\mathrm{atm}} c : \rho'}\ (\text{T-CSub})$$

Fig. 8. The typing rules of the ATM type system $\vdash_{\mathrm{atm}}$.

We extend the simple type system $\vdash_{\mathrm{st}}$ by adding the following rules.

$$\frac{\overline{\Gamma \vdash_{\mathrm{st}} v_i : \sigma_i}}{\Gamma \vdash_{\mathrm{st}} \{\overline{l_i = v_i}\} : \{\overline{l_i : \sigma_i}\}}\ (\text{St-Record}) \quad \frac{\Gamma \vdash_{\mathrm{st}} v : \{\overline{l_i : \sigma_i}\}}{\Gamma \vdash_{\mathrm{st}} v.l_i : \sigma_i}\ (\text{St-Proj})$$

D CPS Transformation for Subtyping Derivations

Figure 10 shows the complete set of CPS transformation rules for subtyping derivations.

$$\frac{}{b \leq b}\ (\text{S-Base}) \quad \frac{\tau_2 \leq \tau_1 \quad \rho_1 \leq \rho_2}{\tau_1 \to \rho_1 \leq \tau_2 \to \rho_2}\ (\text{S-Arr}) \quad \frac{\tau_1 \leq \tau_2}{\tau_1/\Box \leq \tau_2/\Box}\ (\text{S-Pure})$$

$$\frac{\tau_1 \leq \tau_2 \quad \rho_2 \leq \rho_1 \quad \rho_1' \leq \rho_2'}{\tau_1/\rho_1 \Rightarrow \rho_1' \leq \tau_2/\rho_2 \Rightarrow \rho_2'}\ (\text{S-Ipure}) \quad \frac{\tau_1 \leq \tau_2 \quad \rho_1 \leq \rho_2}{\tau_1/\Box \leq \tau_2/\rho_1 \Rightarrow \rho_2}\ (\text{S-Embed})$$

Fig. 9. The subtyping rules of $\vdash_{\mathrm{atm}}$.

$$\begin{aligned}
[\![b \leq b]\!] &= \lambda x.x \\
[\![\tau_1 \to \rho_1 < \tau_2 \to \rho_2]\!] &= \lambda f.\lambda x.[\![\rho_1 \leq \rho_2]\!]@(f@([\![\tau_2 \leq \tau_1]\!]@x)) \\
[\![\tau_1/\Box \leq \tau_2/\Box]\!] &= \lambda x.[\![\tau_1 \leq \tau_2]\!]@x \\
[\![\tau_1/\rho_1 \Rightarrow \rho_1' \leq \tau_2/\rho_2 \Rightarrow \rho_2']\!] & \\
&= \lambda x.\lambda h.\lambda k.[\![\rho_1' \leq \rho_2']\!]@(x@h@\lambda y.[\![\rho_2 \leq \rho_1]\!]@(k@([\![\tau_1 \leq \tau_2]\!]@y))) \\
[\![\tau_1/\Box \leq \tau_2/\rho_1 \Rightarrow \rho_2]\!] &= \lambda x.\lambda h.\lambda k.[\![\rho_1 \leq \rho_2]\!]@(k@([\![\tau_1 \leq \tau_2]\!]@x)
\end{aligned}$$

Fig. 10. The CPS transformation rules for subtyping derivations.

E CPS Transformation for Typing Derivations

Figure 11 shows the complete set of CPS transformation rules for typing derivations.

F Proof of Theorem 5

We first show the basic type soundness property for $\vdash_{\mathrm{atm}}$ by proving the standard preservation and progress properties [44]. Besides sanity checking that $\vdash_{\mathrm{atm}}$ enjoys the usual correctness properties expected for a type system, this is useful for proving Theorem 5 because our CPS transformation is typing-derivation-driven and the fact that typability is preserved allows us to pick a typing derivation for c' to CPS transform it when $\vdash_{\mathrm{atm}} c : \rho$ and $c \to c'$.

Lemma 2 (Progress). *If $\vdash_{\mathrm{atm}} c : \rho$, then either*

- *$c \to c'$ for some c'*
- *$c = \mathtt{return}\ v$ for some v, or*
- *$c = E[op\ v]$ for some E, v, op.*

Proof. Immediate from the definitions of $\vdash_{\mathrm{atm}}$ and $\to$. □

The following is also immediate.

Lemma 3 (Substitution). *If $\Gamma, x : \tau \vdash_{\mathrm{atm}} c : \rho$ and $\Gamma \vdash_{\mathrm{atm}} v : \tau$ then $\Gamma \vdash_{\mathrm{atm}} c[v/x] : \rho$.*

We now show the preservation property.

Lemma 4 (Preservation). *If $\Gamma \vdash_{\mathrm{atm}} c : \rho$ and $c \to c'$, then $\Gamma \vdash_{\mathrm{atm}} c' : \rho$.*

Proof. We prove by induction on the derivation of $c \to c'$. The cases besides (E-Op) are immediate from Lemma 3. For the case (E-Op), we may assume without loss of generality that the derivation $\Gamma \vdash_{\mathrm{atm}} c : \rho$ is of the form

$$\frac{\Gamma \vdash_{\mathrm{atm}} h \quad \Gamma, x : \tau \vdash_{\mathrm{atm}} c'' : \rho' \quad \Gamma \vdash_{\mathrm{atm}} E[op\ v] : \tau/\rho' \Rightarrow \rho}{\Gamma \vdash_{\mathrm{atm}} \mathtt{with}\ h\ \mathtt{handle}\ E[op\ v] : \rho}\ (\text{T-Han})$$

where $\mathtt{return}\ x = c'' \in h$ and the subderivation $\Gamma \vdash_{\mathrm{atm}} E[op\ v] : \tau/\rho' \Rightarrow \rho$ is of the form

$$\frac{\begin{array}{c}\mathcal{D}\\ \vdots\end{array}}{\Gamma \vdash_{\mathrm{atm}} E[op\ v] : \tau/\rho' \Rightarrow \rho}$$

containing a subderivation $\mathcal{D}$ of the form

$$\frac{\Gamma(op) = \tau_2 \to \tau'/\rho_2 \Rightarrow \rho \quad \Gamma \vdash_{\mathrm{atm}} v : \tau_2}{\Gamma \vdash_{\mathrm{atm}} op\ v : \tau'/\rho_2 \Rightarrow \rho}\ (\text{T-Op})$$

By replacing this $\mathcal{D}$, by the following derivation $\mathcal{D}'$

$$\frac{\Gamma, y : \tau_2 \vdash_{\mathrm{atm}} \mathtt{return}\ y : \tau_2/\Box \quad \tau_2/\Box \leq \tau_2/\rho_2 \Rightarrow \rho_2}{\Gamma, y : \tau_2 \vdash_{\mathrm{atm}} \mathtt{return}\ y : \tau_2/\rho_2 \Rightarrow \rho_2}\ (\text{T-Csub})$$

we obtain the derivation

$$\frac{\Gamma' \vdash_{\mathrm{atm}} h \quad \Gamma', x : \tau \vdash_{\mathrm{atm}} c'' : \rho' \quad \Gamma' \vdash_{\mathrm{atm}} E[\mathtt{return}\ y] : \tau/\rho' \Rightarrow \rho_2}{\Gamma' \vdash_{\mathrm{atm}} \mathtt{with}\ h\ \mathtt{handle}\ E[\mathtt{return}\ y] : \rho_2}\ (\text{T-Han})$$

where $\Gamma' = \Gamma, y : \tau_2$. Therefore, by (T-Hdlr) and Lemma 3, we have

$$\Gamma \vdash_{\mathrm{atm}} c[v/x, \lambda y.\mathtt{with}\ h\ \mathtt{handle}\ E[\mathtt{return}\ y]/k] : \rho$$

□

We obtain the type soundness property as a corollary of the progress and the preservation properties.

Corollary 1 (Type Soundness). *If c is $\vdash_{\mathrm{atm}}$-typable and $c \to^* c'$, then either*

- *$c' \to c''$ for some $\vdash_{\mathrm{atm}}$-typable c''*
- *$c' = \mathtt{return}\ v$ for some v, or*
- *$c' = E[op\ v]$ for some E, v, op.*

The following lemma states that substitution and CPS transformation commute.

Lemma 5. *Let $\mathcal{D}_c$ be a typing derivation of $\Gamma, x : \tau \vdash_{\mathrm{atm}} c : \rho$, and $\mathcal{D}_v$ be a typing derivation of $\Gamma \vdash_{\mathrm{atm}} v : \tau$. Then, $[\![\mathcal{D}_c[\mathcal{D}_v/x]]\!] = [\![\mathcal{D}_c]\!][[\![\mathcal{D}_v]\!]/x]$ where $\mathcal{D}_c[\mathcal{D}_v/x]$ denotes the derivation obtained by replacing occurrences of $\Gamma, x : \tau \vdash_{\mathrm{atm}} x : \tau$ by $\mathcal{D}_v$.*

Proof. By induction on the derivation $\mathcal{D}_c$ □

Next we prove that one step of the source program evaluation can be simulated by the transformed program.

Lemma 6 (One-Step Forward Simulation). *Let $\mathcal{D}$ be a derivation of $\Gamma \vdash_{\mathrm{atm}} c : \rho$ and $c \to c'$. If ρ is pure, then there is a derivation $\mathcal{D}'$ of $\Gamma \vdash_{\mathrm{atm}} c' : \rho$ such that $[\![\mathcal{D}]\!] \to^+ [\![\mathcal{D}']\!]$. Otherwise, ρ is effectful, and there is a derivation $\mathcal{D}'$ of $\Gamma \vdash_{\mathrm{atm}} c' : \rho$ such that $[\![\mathcal{D}]\!]@v_h@v_k \to^+ [\![\mathcal{D}']\!]@v_h@v_k$ for any values v_h and v_k.*

Proof. Let us fix a derivation $\mathcal{D}'$ for c' constructed by Lemma 4. We prove by simultaneous induction on the derivation of $c \to c'$ and the derivation $\mathcal{D}$.

First, we consider the case the root of $\mathcal{D}$ is an instance of (T-CSub). $\mathcal{D}$ must be of the form

$$\frac{\Gamma \vdash_{\mathrm{atm}} c : \rho' \quad \rho' \leq \rho}{\Gamma \vdash_{\mathrm{atm}} c : \rho} \text{ (T-CSub)}$$

Therefore, $[\![\mathcal{D}]\!] = [\![\rho' \leq \rho]\!]@[\![\Gamma \vdash_{\mathrm{atm}} c : \rho']\!]$ and $[\![\mathcal{D}']\!] = [\![\rho' \leq \rho]\!]@[\![\Gamma \vdash_{\mathrm{atm}} c' : \rho']\!]$. If ρ is pure, then we have

$$[\![\mathcal{D}]\!] = [\![\rho' \leq \rho]\!]@[\![\Gamma \vdash_{\mathrm{atm}} c : \rho']\!] \to^+ [\![\rho' \leq \rho]\!]@[\![\Gamma \vdash_{\mathrm{atm}} c' : \rho']\!] = [\![\mathcal{D}']\!]$$

Otherwise, ρ is effectful and is of the form $\tau/\rho_1 \Rightarrow \rho_2$ for some τ, ρ_1, and ρ_2. If ρ' is a pure computation type of the form $\tau'/\Box$, then

$$\begin{aligned}
[\![\mathcal{D}]\!]@v_h@v_k &= [\![\rho' \leq \rho]\!]@[\![\Gamma \vdash_{\mathrm{atm}} c : \rho']\!]@v_h@v_k \\
&= [\![\rho_1 \leq \rho_2]\!]@(v_k\ ([\![\tau' \leq \tau]\!]@[\![\Gamma \vdash_{\mathrm{atm}} c : \rho']\!])) \\
&\to^+ [\![\rho_1 \leq \rho_2]\!]@(v_k\ ([\![\tau' \leq \tau]\!]@[\![\Gamma \vdash_{\mathrm{atm}} c' : \rho']\!])) \\
&= [\![\rho' \leq \rho]\!]@[\![\Gamma \vdash_{\mathrm{atm}} c' : \rho']\!]@v_h@v_k \\
&= [\![\mathcal{D}']\!]@v_h@v_k
\end{aligned}$$

Otherwise, ρ' is also effectful and is of the form $\tau'/\rho'_1 \Rightarrow \rho'_2$. Then,

$$\begin{aligned}
[\![\mathcal{D}]\!]@v_h@v_k &= [\![\rho' \leq \rho]\!]@[\![\Gamma \vdash_{\mathrm{atm}} c : \rho']\!]@v_h@v_k \\
&= [\![\rho'_2 \leq \rho_2]\!]@([\![\Gamma \vdash_{\mathrm{atm}} c : \rho']\!]@v_h@\lambda y.[\![\rho_1 \leq \rho'_1]\!]@(k@([\![\tau \leq \tau']\!]@y)) \\
&\to^+ [\![\rho'_2 \leq \rho_2]\!]@([\![\Gamma \vdash_{\mathrm{atm}} c' : \rho']\!]@v_h@\lambda y.[\![\rho_1 \leq \rho'_1]\!]@(k@([\![\tau \leq \tau']\!]@y)) \\
&= [\![\rho' \leq \rho]\!]@[\![\Gamma \vdash_{\mathrm{atm}} c' : \rho']\!]@v_h@v_k \\
&= [\![\mathcal{D}']\!]@v_h@v_k
\end{aligned}$$

In what follows, we assume that the last rule used in $\mathcal{D}$ is not (T-CSub). We prove by case analysis on the last rule of $c \to c'$.

Suppose that it is (E-Let). Then, $c \to c'$ is of the form

$$\frac{c_1 \to c'_1}{\mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 \to \mathtt{let}\ x = c'_1\ \mathtt{in}\ c_2} \text{ (E-Let)}$$

If ρ is pure and is of the form $\tau/\Box$, then $\mathcal{D}$ must be of the form

$$\frac{\Gamma \vdash_{\mathrm{atm}} c_1 : \tau_1/\Box \quad \Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau/\Box}{\Gamma \vdash_{\mathrm{atm}} \mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 : \tau/\Box} \text{ (T-LetP)}$$

Therefore,

$$\begin{array}{rl} [\![\mathcal{D}]\!] = & (\lambda x.[\![\Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau/\square]\!])\ [\![\Gamma \vdash_{\mathrm{atm}} c_1 : \tau_1/\square]\!] \\ \to^+ & (\lambda x.[\![\Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau/\square]\!])\ [\![\Gamma \vdash_{\mathrm{atm}} c_1' : \tau_1/\square]\!] \\ = & [\![\mathcal{D}']\!] \end{array}$$

If ρ is effectful and is of the form $\tau/\rho_1 \Rightarrow \rho_2$, then $\mathcal{D}$ must be of the form

$$\frac{\Gamma \vdash_{\mathrm{atm}} c_1 : \tau_1/\rho' \Rightarrow \rho_2 \quad \Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau/\rho_1 \Rightarrow \rho'}{\Gamma \vdash_{\mathrm{atm}} \mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 : \tau/\rho_1 \Rightarrow \rho_2}\ (\textsc{T-LetIp})$$

Therefore,

$$\begin{array}{ll} & [\![\mathcal{D}]\!]@v_h@v_k \\ = & [\![\Gamma \vdash_{\mathrm{atm}} c_1 : \tau_1/\rho' \Rightarrow \rho_2]\!]@v_h@(\lambda x.[\![\Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau/\rho_1 \Rightarrow \rho']\!]@v_h@v_k) \\ \to^+ & [\![\Gamma \vdash_{\mathrm{atm}} c_1' : \tau_1/\rho' \Rightarrow \rho_2]\!]@v_h@(\lambda x.[\![\Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau/\rho_1 \Rightarrow \rho']\!]@v_h@v_k) \\ = & [\![\mathcal{D}']\!]@v_h@v_k \end{array}$$

Next, suppose that the last rule is (E-Ret). Then, $c \to c'$ must be of the form

$$\frac{}{\mathtt{let}\ x = \mathtt{return}\ v\ \mathtt{in}\ c_2 \to c_2[v/x]}$$

If ρ is a pure type of the form $\tau/\square$ then $\mathcal{D}$ must be of the form

$$\frac{\dfrac{\Gamma \vdash_{\mathrm{atm}} v : \tau_1}{\Gamma \vdash_{\mathrm{atm}} \mathtt{return}\ v : \tau_1/\square} \quad \Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau/\square}{\Gamma \vdash_{\mathrm{atm}} \mathtt{let}\ x = \mathtt{return}\ v\ \mathtt{in}\ c_2 : \tau/\square}$$

Therefore,

$$\begin{array}{rl} [\![\mathcal{D}]\!] = & (\lambda x.[\![\Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau/\square]\!])\ [\![\Gamma \vdash_{\mathrm{atm}} v : \tau_1]\!] \\ \to & [\![\Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau/\square]\!][[\![\Gamma \vdash_{\mathrm{atm}} v : \tau_1]\!]/x] \\ = & [\![\mathcal{D}']\!] \end{array}$$

where the last equation follows by Lemma 5. If ρ is an effectful type of the form $\tau/\rho_1 \Rightarrow \rho_2$ then $\mathcal{D}$ must be of the form

$$\frac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2}{\Gamma \vdash_{\mathrm{atm}} \mathtt{return}\ v : \tau_1/\rho' \Rightarrow \rho_2} \quad \Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau/\rho_1 \Rightarrow \rho'}{\Gamma \vdash_{\mathrm{atm}} \mathtt{let}\ x = \mathtt{return}\ v\ \mathtt{in}\ c_2 : \tau/\rho_1 \Rightarrow \rho_2}$$

where $\mathcal{D}_1$ and $\mathcal{D}_2$ are as follows

$$\frac{\Gamma \vdash_{\mathrm{atm}} v : \tau_1}{\Gamma \vdash_{\mathrm{atm}} \mathtt{return}\ v : \tau_1/\square} \qquad \frac{\rho' \le \rho_2}{\tau_1/\square \le \tau_1/\rho' \Rightarrow \rho_2}$$

Therefore,

$$\begin{array}{ll} & [\![\mathcal{D}]\!]@v_h@v_k \\ = & [\![\rho' \le \rho_2]\!]@((\lambda x.[\![\Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau/\rho_1 \Rightarrow \rho']\!]@v_h@v_k)\ [\![\Gamma \vdash_{\mathrm{atm}} v : \tau_1]\!]) \\ \to & [\![\rho' \le \rho_2]\!]@([\![\Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau/\rho_1 \Rightarrow \rho']\!][[\![\Gamma \vdash_{\mathrm{atm}} v : \tau_1]\!]/x]@v_h@v_k) \\ = & [\![\mathcal{D}']\!]@v_h@v_k \end{array}$$

where the last equation follows by Lemma 5 again. The other cases can be similarly shown by applying Lemma 5. □

The forward direction of the simulation theorem, that is, item 1 of Theorem 5, follows by repeated applications of Lemma 6.

Next we prove the backward direction, that is, item 2 of Theorem 5. Because c is $\vdash_{\mathrm{atm}}$-typable, by Corollary 1, the evaluation of c either (1) diverges, (2) gets stuck by reaching some $E[op\ v]$, or (3) halts by returning a value. Note that the evaluation relation $\to$ is deterministic. If (1) is true, then by the forward direction of the theorem that we have just shown, it must be the case that the evaluation of $[\![\vdash_{\mathrm{atm}} c : \tau/\square]\!]$ also diverges. Next, suppose that (2) is true. Then, by Lemma 4, it must be the case that $\vdash_{\mathrm{atm}} E[op\ v] : \tau/\square$, but such a typing is not possible because $E[op\ v]$ can only be given an effectful computation type. Finally, suppose that (3) is true. Let $c \to^* \mathtt{return}\ v$. It suffices to show that $[\![\vdash_{\mathrm{atm}} v : \tau]\!] = v'$ where $[\![\vdash_{\mathrm{atm}} c : \tau/\square]\!] \to^* \mathtt{return}\ v'$. But, $[\![\vdash_{\mathrm{atm}} c : \tau/\square]\!] \to^* [\![\vdash_{\mathrm{atm}} \mathtt{return}\ v : \tau/\square]\!]$ by Lemma 6. Therefore, by the determinism of the evaluation relation, it must be the case that $[\![\vdash_{\mathrm{atm}} v : \tau]\!] = v'$. This complete the proof of Theorem 5.

G Proof of Theorem 7

We note that the proof of the theorem, as well as the construction of the class of programs MM, is an adaptation of the proof of the undecidability of the reachability problem for finitary PCF extended with exceptions given in [17]. Namely, we adapt their proof to our setting by implementing the exceptions in their programs by effect handlers, following the usual approach of implementing exceptions by effect handlers [35].[9]

First, we recall 2-register Minsky machines [25]. A *2-register Minsky machine* is a tuple $M = (Q, \delta, q_0)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, δ is the transition function which maps each state $q \in Q$ to the set of instructions of the following three forms:

Inc r_i; **goto** q' Increment register r_i ($i \in \{0, 1\}$) and go to state q' ($q' \in Q$).
If $r_i = 0$ **then goto** q_1 **else dec** r_i; **goto** q_2 Check the value of register r_i ($i \in \{0, 1\}$). If the value is 0 then go to state q_1 ($q_1 \in Q$). Otherwise, decrement r_i and go to state q_2 ($q_2 \in Q$).
Halt Halt the machine.

A *configuration* of the machine is a triple (q, n_0, n_1) where $q \in Q$ and n_0 and n_1 are non-negative integers denoting the values of the two registers. The computation of the machine starts from the initial configuration $(q_0, 0, 0)$ and terminates when the Halt instruction is reached. The *halting problem* for 2-counter Minsky

[9] The paper [17] contains a proof that reachability for finitary PCF extended with effect handlers is undecidable (with a non-ATM type system), but it uses a different class of programs because the restriction on operation parameters mentioned in Section 3.4 prevents them from using MM.

machines is to decide if the computation of a given 2-counter Minsky machine terminates.

Theorem 8 ([25]). *The halting problem for 2-counter Minsky machines is undecidable.*

It is easy to see that each program in MM^n implements an n+1-state 2-register Minsky machine by noticing that the recursive functions $f_0, \ldots, f_n$ represent the n+1 states with f_0 in particular representing the initial state q_0, each $c_{inc}^{i,j}$ implements the increment instruction that increments the register r_i and go to the state represented by f_j, and each $c_{dec}^{i,j,m}$ implements the check-and-decrement instruction that checks the register r_i, goes to the state represented by f_j if it is 0 and otherwise decrements r_i and goes to the state represented by f_m. The halt instruction is implemented by $()$. Therefore, a program in MM returns $\top$ iff the corresponding 2-register Minsky machine halts. Thus, the halting problem for 2-counter Minsky machines is reduced to the reachability problem for MM. This completes the proof of Theorem 7.

$$[\![\Gamma \vdash_{\mathrm{atm}} () : \mathtt{unit}]\!] = () \qquad \left[\!\!\left[\frac{v \in \{\top, \bot\}}{\Gamma \vdash_{\mathrm{atm}} v : \mathtt{bool}}\right]\!\!\right] = v \qquad \left[\!\!\left[\frac{x : \tau \in \Gamma}{\Gamma \vdash_{\mathrm{atm}} x : \tau}\right]\!\!\right] = x$$

$$\left[\!\!\left[\frac{\Gamma, x : \tau \vdash_{\mathrm{atm}} c : \rho}{\Gamma \vdash_{\mathrm{atm}} \lambda x.c : \tau \to \rho}\right]\!\!\right] = \lambda x.\, [\![\Gamma, x : \tau \vdash_{\mathrm{atm}} c : \rho]\!]$$

$$\left[\!\!\left[\frac{\Gamma, x : \tau \vdash_{\mathrm{atm}} v : \tau}{\Gamma \vdash_{\mathrm{atm}} \mathtt{rec}\ x = v : \tau}\right]\!\!\right] = \mathtt{rec}\ x = [\![\Gamma, x : \tau \vdash_{\mathrm{atm}} v : \tau]\!]$$

$$\left[\!\!\left[\frac{\Gamma \vdash_{\mathrm{atm}} v : \mathtt{bool} \quad \Gamma \vdash_{\mathrm{atm}} c_1 : \rho \quad \Gamma \vdash_{\mathrm{atm}} c_2 : \rho}{\Gamma \vdash_{\mathrm{atm}} \mathtt{if}\ v\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2}\right]\!\!\right] = \mathtt{if}\ [\![\Gamma \vdash_{\mathrm{atm}} v : \mathtt{bool}]\!]\ \mathtt{then}\ [\![\Gamma \vdash_{\mathrm{atm}} c_1 : \rho]\!]\ \mathtt{else}\ [\![\Gamma \vdash_{\mathrm{atm}} c_2 : \rho]\!]$$

$$\left[\!\!\left[\frac{\Gamma \vdash_{\mathrm{atm}} v_1 : \tau \to \rho \quad \Gamma \vdash_{\mathrm{atm}} v_2 : \tau}{\Gamma \vdash_{\mathrm{atm}} v_1\ v_2 : \rho}\right]\!\!\right] = [\![\Gamma \vdash_{\mathrm{atm}} v_1 : \tau \to \rho]\!]\ [\![\Gamma \vdash_{\mathrm{atm}} v_2 : \tau]\!]$$

$$\left[\!\!\left[\frac{\Gamma \vdash_{\mathrm{atm}} c_1 : \tau_1/\square \quad \Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau_2/\square}{\Gamma \vdash_{\mathrm{atm}} \mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 : \tau_2/\square}\right]\!\!\right] = (\lambda x.\, [\![\Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau_2/\square]\!])\ [\![\Gamma \vdash_{\mathrm{atm}} c_1 : \tau_1/\square]\!]$$

$$\left[\!\!\left[\frac{\Gamma \vdash_{\mathrm{atm}} c_1 : \tau_1/\rho_1 \Rightarrow \rho_1' \quad \Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau_2/\rho_2 \Rightarrow \rho_1}{\Gamma \vdash_{\mathrm{atm}} \mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 : \tau_2/\rho_2 \Rightarrow \rho_1'}\right]\!\!\right] = \lambda h.\lambda k.\, [\![\Gamma \vdash_{\mathrm{atm}} c_1 : \tau_1/\rho_1 \Rightarrow \rho_1']\!]\ @h@\ (\lambda x.\, [\![\Gamma, x : \tau_1 \vdash_{\mathrm{atm}} c_2 : \tau_2/\rho_2 \Rightarrow \rho_1]\!]\ @h@k)$$

$$\left[\!\!\left[\frac{\Gamma \vdash_{\mathrm{atm}} v : \tau}{\Gamma \vdash_{\mathrm{atm}} \mathtt{return}\ v : \tau/\square}\right]\!\!\right] = [\![\Gamma \vdash_{\mathrm{atm}} v : \tau]\!]$$

$$\left[\!\!\left[\frac{\Sigma(op) = \tau \to \tau'/\rho_1 \Rightarrow \rho_2 \quad \Gamma \vdash_{\mathrm{atm}} v : \tau}{\Gamma \vdash_{\mathrm{atm}} op\ v : \tau'/\rho_1 \Rightarrow \rho_2}\right]\!\!\right] = \lambda h.\lambda k.h.op\ [\![\Gamma \vdash_{\mathrm{atm}} v : \tau]\!]\ k$$

$$\left[\!\!\left[\frac{\overline{\Sigma(op_i) = \tau_i \to \tau_i'/\rho_i \Rightarrow \rho_i'} \quad \overline{\Gamma, x_i : \tau_i, k_i : \tau_i' \to \rho_i \vdash_{\mathrm{atm}} c_i : \rho_i'}}{\Gamma \vdash_{\mathrm{atm}} \{\mathtt{return}\ x = c, \overline{op_i\ x_i\ k_i = c_i}\}}\right]\!\!\right] = \left\{\overline{op_i = \lambda x_i.\lambda k_i.[\![\Gamma, x_i : \tau_i, k_i : \tau_i' \to \rho_i \vdash_{\mathrm{atm}} c_i : \rho_i']\!]}\right\}$$

$$\left[\!\!\left[\frac{\Gamma \vdash_{\mathrm{atm}} h \quad \Gamma \vdash_{\mathrm{atm}} c : \tau/\rho \Rightarrow \rho' \quad \Gamma, x : \tau \vdash_{\mathrm{atm}} c' : \rho \quad \mathtt{return}\ x = c' \in h}{\Gamma \vdash_{\mathrm{atm}} \mathtt{with}\ h\ \mathtt{handle}\ c : \rho'}\right]\!\!\right] = [\![\Gamma \vdash_{\mathrm{atm}} c : \tau/\rho \Rightarrow \rho']\!] \mathbin{@} [\![\Gamma \vdash_{\mathrm{atm}} h]\!] \mathbin{@} \lambda x.\, [\![\Gamma, x : \tau \vdash_{\mathrm{atm}} c' : \rho]\!]$$

$$\left[\!\!\left[\frac{\Gamma \vdash_{\mathrm{atm}} c : \rho \quad \rho \leq \rho'}{\Gamma \vdash_{\mathrm{atm}} c : \rho'}\right]\!\!\right] = [\![\rho \leq \rho']\!] \mathbin{@} [\![\Gamma \vdash_{\mathrm{atm}} c : \rho]\!]$$

$$\left[\!\!\left[\frac{\Gamma \vdash_{\mathrm{atm}} v : \tau \quad \tau \leq \tau'}{\Gamma \vdash_{\mathrm{atm}} v : \tau'}\right]\!\!\right] = [\![\tau \leq \tau']\!] \mathbin{@} [\![\Gamma \vdash_{\mathrm{atm}} v : \tau]\!]$$

Fig. 11. The CPS transformation rules for typing derivations.