m-CFA Exhibits Perfect Stack Precision

Kimball Germane[1 [0000−0003−4903−5645]]

Brigham Young University, Provo UT 84601, USA kimball@cs.byu.edu

Abstract. m-CFA is a hierarchy of control-flow analyses (CFA) formulated as abstract abstract machines and designed to exhibit polynomial time complexity while remaining usefully precise. The *Pushdown for Free* technique (P4F) prescribes a continuation allocator which induces *perfect stack precision* wherein each function invocation returns to only its call. Unfortunately, it is difficult to apply P4F to m-CFA as P4F is developed in an ANF setting but m-CFA is formulated in a CPS setting. In this paper, we recall that ANF corresponds to a CPS sublanguage without non-local control and show that m-CFA behaves identically on both. With an ANF-based m-CFA in hand, we turn to applying P4F only to discover that it already follows the prescription. In other words, m-CFA has always had perfect stack precision, a characteristic neither intended nor recognized, at its development or since. In addition to being surprising, we discuss how this result allows a spectrum of non-local control constructs to be supported more easily and with more precision than previous techniques.

Keywords: Static analysis · Control-flow analysis · Abstract interpretation

1 Introduction

A flow analysis of a functional program (i.e. control-flow analysis or CFA) computes, for each call $(f\,e)$, the set of (closures over) λs which flow to f (i.e. to which f may evaluate) and, for each function $\lambda x.e$, the set of enclosed λs which flow to x (i.e. to which x may be bound) [14]. Perhaps the most prevalent flow analysis is Shivers's k-CFA [16], a hierarchy of analyses in which the CFA at level k qualifies the analysis of each expression by the last-k call sites encountered during abstract evaluation. For instance, 0CFA does not qualify the analysis of expressions at all, and is thus context-insensitive; in contrast, 1CFA uses the most-recent call site to distinguish the analysis of otherwise-identical evaluation. To illustrate each, consider the program to the right, adapted from Gilray *et al.* [6], which we will use as a running example. The function `id` is called once at each of two sites with different arguments and, as a consequence, the analysis will bind `x` twice. A 0CFA analysis will conflate these two bindings so that each reference to `x` produces the values of *both* arguments. A 1CFA analysis, on the other hand, will qualify each binding by the most-recent call

```
(let* ([id (lambda (x) x)]
       [y (id 10)]
       [z (id 12)])
  (+ y z))
```

site, `(id 10)` and `(id 12)` respectively, so that references to it access only the values so qualified.

Since Shivers introduced k-CFA, techniques have been developed to improve its precision [12], its power [12, 11, 21], and its engineerability [19]. In this paper, we recall and reconcile two concurrent improvements, the development of m-CFA and the development of stack-precise CFA.

m-CFA [13] emerged from a kind of paradox: when k-CFA is applied to a functional language, its complexity is exponential (for $k > 0$); when k-CFA is applied to an object-oriented (OO) language, however, its complexity is polynomial. The discrepancy arises from the different ways in which environments are created in each setting. In a functional program, environments are created implicitly when a λ is encountered whereas, in an OO program, environments are created as part of explicit constructor invocation using `new`. Might *et al.* resolve this discrepancy to obtain a context-sensitive CFA hierarchy, m-CFA, with polynomial time complexity.

Stack-precise CFA emerged from the desire for a better model of control flow in functional languages. For two decades, CFAs modelled control flow as a finite state machine (FSM), a directed graph of control states connected by control transitions. While this model can be produced by relatively-simple workset algorithms, it cannot precisely capture the control behavior of higher-order programs whose execution is facilitated by a stack. Without precisely modeling the stack, it is impossible to capture the call–return behavior of programs with full precision, and FSM-producing CFAs routinely lose track of which particular caller to which a given call should return. The example program illustrates this well: although a 1CFA produces the expected value for each dynamic reference to `x`, a 1CFA without a precise stack model may associate both returns from `id` to each caller. In this case, `y` and `z` are each bound to both dynamic values of `x`, and the analysis calculates a result set $\{10 + 10, 10 + 12 = 12 + 10, 12 + 12\}$.

In the same year as m-CFA's presentation, Vardoulakis and Shivers [20] presented CFA2, a "context-free approach to control-flow analysis", which models control flow using a pushdown system. Using a pushdown system, rather than an FSM, allows CFA2 to precisely model the stack and perfectly associate each return to its corresponding call. Unfortunately, CFA2's summarization algorithm is substantially more complex than an FSM-producing workset algorithm and must be significantly modified to accommodate additional control features [22]. Moreover, its computational complexity is exponential, despite CFA2 not exhibiting call-site sensitivity à la k-CFA. Follow-on work produced stack-precise CFAs corresponding to FSM CFAs whose context-insensitive instances had polynomial complexity [10, 7], but the techniques still imposed polynomial overhead and, in some cases, employed similarly-intricate summarization algorithms.

Somewhat surprisingly, Gilray *et al.* [6] discovered a technique to transform an FSM-based CFA into a stack-precise CFA "for free" in two senses: first, the technique prescribes a particular continuation allocator but requires no modification to the CFA, so it is free in terms of implementation effort; second, the allocator imposes only a constant factor overhead to running time above the

CFA's, and so it is free in terms of computational complexity. Following the authors, we refer to this as the *pushdown for free* technique, abbreviated *P4F*.

Naturally, we would like to apply P4F to m-CFA to get the best of both worlds: a (1) polynomial-time, (2) stack-precise CFA hierarchy that (3) admits a straightforward workset-based implementation. Applying P4F requires care, however, because m-CFA is defined in terms of a CPS language but P4F is demonstrated in an ANF setting [4], and a naïve port will not necessarily result in the same analysis [15].

In this paper, we reformulate m-CFA so as to be able to directly apply P4F. After reviewing m-CFA (§2), we identify a subset of its CPS language free from non-local control (§3) and specialize a formulation of m-CFA to it (§4). We then translate this subset language to ANF (§5), formulate m-CFA for it (§6), and show that it is the same analysis as the CPS-based one (§7). Having arrived in ANF, we review P4F (§8). We observe that ANF-based CFA already uses it and show that it is indeed stack-precise (§9). We conclude by discussing ramifications of the corollary that, within the subset language, *m-CFA is and always has been stack-precise* (§10).

2 m-CFA

Might *et al.* [13] developed m-CFA in response to the paradox that, when formulated in an object-oriented (OO) setting, k-CFA [16] exhibits polynomial time complexity but, when formulated in a functional setting, exhibits exponential time complexity. After ensuring that k-CFA is implemented faithfully in both settings, Might *et al.* pinpoint environment construction to be the key distinction: in functional languages, environment bindings are captured implicitly within closures when lambda expressions are evaluated; in contrast, programmers explicitly pass data to constructors in OO languages when creating new objects. This difference leads to an exponential number of possible environments in the former case and a polynomial number in the latter, explaining the discrepancy.

Might *et al.* resolve this paradox by modifying k-CFA to produce only a polynomial number of environments by flattening the environment structure. To support this structure, their modified analysis explicitly copies bindings from old environments to new at each step, mimicking the manual construction that programmers carry out in OO programs. However, they observe that this rebinding policy leads to a precision decrease in typical programs, which is visible in the program to the right. In 1CFA, the bindings of `x` in `f` *before* the call to `log` are distinguished by the `f`'s caller, it being the most-recent call site. *After* the call to `log`, however, the most-recent call site is this call to `log`, or its last inner call, regardless of `f`'s caller. Consequently, rebinding `x` from the former environment to the latter combines bindings from distinct callers, jettisoning precision. Rather than revert the policy to avoid a precision decrease, Might *et al.* manage the context abstraction differently. Instead of qualifying evaluation with the last-k call sites, they devise an approach which qualifies it with the top-m stack

```
(define (f x)
  (log "f call")
  (g x))
```

frames. The form of the context remains the same—a sequence of call sites—but its construction and consequent effect on the analysis differs. The resulting analysis, m-CFA, is characterized by both its rebinding policy and its context abstraction.

m-CFA is defined over a CPS language, in which all control is effected through function calls, in terms of a small-step abstract machine. We reproduce its formalism in Figure 1, remaining vague about the details of the CPS language until §3. A machine state $\tilde{\varsigma}$ is a tuple of a (CPS) call, environment, and store. A store

$$\tilde{\varsigma} \in \tilde{\Sigma} = \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \qquad \hat{\sigma} \in \widehat{Store} = \widehat{Addr} \to \hat{D}$$
$$\hat{d} \in \hat{D} = \mathcal{P}(\widehat{Clo} + \{\mathsf{halt}\}) \qquad \widehat{clo} \in \widehat{Clo} = \mathsf{Lam} \times \widehat{Env}$$
$$\hat{a} \in \widehat{Addr} = \mathsf{Var} \times \widehat{Env} \qquad \hat{\rho} \in \widehat{Env} = \mathsf{ULab}^{\leq m}$$

$$\Rightarrow_{\tilde{\Sigma}} \subseteq \tilde{\Sigma} \times \tilde{\Sigma}$$

$$(call, \hat{\rho}, \hat{\sigma}) \Rightarrow_{\tilde{\Sigma}} (call', \hat{\rho}'', \hat{\sigma}') \text{ where } call = [\![(f\, e_1 \ldots e_n)^\ell]\!] \text{ and}$$
$$(lam, \hat{\rho}') \in \hat{\mathcal{E}}(f, \hat{\rho}, \hat{\sigma}) \qquad \hat{d}_i = \hat{\mathcal{E}}(f, \hat{\rho}, \hat{\sigma})$$
$$lam = [\![(\lambda\,(v_1 \ldots v_n)\, call')]\!] \qquad \hat{a}_{x_j} = (x_j, \hat{\rho}'')$$
$$\hat{\rho}'' = \widehat{new}(\ell, \hat{\rho}, lam, \hat{\rho}') \qquad \hat{a}_{v_i} = (v_i, \hat{\rho}'')$$
$$\{x_1, \ldots, x_m\} = free(lam) \qquad \hat{d}'_j = \hat{\sigma}(x_j, \hat{\rho}')$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_{v_i} \mapsto \hat{d}_i] \sqcup [\hat{a}_{x_j} \mapsto \hat{d}'_j]$$

Fig. 1. m-CFA state transition relation

$\hat{\sigma}$ maps addresses to denotable values. A denotable value $\hat{d}$ is a set of closures, each of which is a pair of a λ expression and an environment. An address $\hat{a}$ is a pair of a variable and an environment. An environment $\hat{\rho}$ is a sequence of call site labels up to length m, which is a parameter to the analysis. These labels are drawn from ULab which we define shortly.

Because of the uniformity of CPS, the machine state transition $\Rightarrow_{\tilde{\Sigma}}$ can be characterized by a single rule: a machine step transitions control from a call to the body of its operator, which is also a call. In CPS, each argument to a call is trivial, and its value is provided by $\hat{\mathcal{E}} : \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \to \hat{D}$.

$$\hat{\mathcal{E}}(x, \hat{\rho}, \hat{\sigma}) = \hat{\sigma}(x, \hat{\rho}) \qquad \hat{\mathcal{E}}(lam, \hat{\rho}, \hat{\sigma}) = \{(lam, \hat{\rho})\}$$

The $\widehat{new}$ metafunction determines the destination environment as a function of the current call and its environment and the operator λ and its environment.

$$\widehat{new}(\ell, \hat{\rho}, lam, \hat{\rho}') = \begin{cases} \lfloor \ell :: \hat{\rho} \rfloor_m & lam \text{ is a procedure} \\ \hat{\rho}' & lam \text{ is a continuation} \end{cases}$$

If the call is the application of a procedure, the destination environment is derived from the source environment by prepending the label of the call being

performed and limiting the environment sequence to m calls overall. If the call is the application of a continuation, its environment is used as the destination environment. In the calculated environment m-CFA installs two distinct sets of bindings: first, the values of each parameter; second, the values of each free variable in the operator λ, resolved in the closure environment.

m-CFA's system space $\tilde{\Xi}$ factors the store from machine states so that an analysis consists of a single, global store and a set $\tilde{R}$ of store-less *configurations*.

$$\tilde{\xi} \in \tilde{\Xi} = \tilde{R} \times \widehat{Store} \qquad \tilde{r} \in \tilde{R} = \mathcal{P}(\tilde{C}) \qquad \tilde{c} \in \tilde{C} = \mathsf{Call} \times \widehat{Env}$$

An analysis is the least fixed point of the total monotonic function $\Rightarrow_{\tilde{\Xi}} \colon \tilde{\Xi} \to \tilde{\Xi}$.

$$(\tilde{C}, \hat{\sigma}) \Rightarrow_{\tilde{\Xi}} (\tilde{C}_0 \cup \tilde{C} \cup \tilde{C}', \hat{\sigma}_0 \sqcup \hat{\sigma}')$$

where $\tilde{C}_0 = \{(call, \langle\rangle)\}$ and $\hat{\sigma}_0 = [(k, \langle\rangle) \mapsto \{\mathsf{halt}\}]$ for program $(\lambda\,(k)\,call)$, and

$$\tilde{S}' = \{\tilde{\varsigma}' : \tilde{c} \in \tilde{C} \text{ and } (\tilde{c}, \hat{\sigma}) \Rightarrow_{\tilde{\Sigma}} \tilde{\varsigma}'\} \quad \tilde{C}' = \{\tilde{c} : (\tilde{c}, \hat{\sigma}) \in \tilde{S}'\} \quad \hat{\sigma}' = \bigsqcup_{(\tilde{c}, \hat{\sigma}) \in \tilde{S}'} \hat{\sigma}.$$

(Here $((call, \hat{\rho}), \hat{\sigma})$ is treated as $(call, \hat{\rho}, \hat{\sigma})$ for convenience.) The definition uses the standard semilattice definition for the store:

$$\bot_{\hat{\sigma}} = \lambda \hat{a}.\emptyset \quad \hat{\sigma}_0 \sqcup \hat{\sigma}_1 = \lambda \hat{a}.\hat{\sigma}_0(\hat{a}) \cup \hat{\sigma}_1(\hat{a}) \quad \hat{\sigma}_0 \sqsubseteq \hat{\sigma}_1 \iff \forall \hat{a} \in \widehat{Addr}.\hat{\sigma}_0(\hat{a}) \subseteq \hat{\sigma}_1(\hat{a})$$

3 CPS and Restricted CPS

Program processors, such as compilers and analyzers, often desugar a rich surface language into more uniform intermediate representation (IR). Modern languages a rich in control constructs, such as branching, function call, early return, and coroutines, and continuation-passing style (CPS) IRs, which express all control transfer via function call, capably regularize such features. m-CFA is defined over a quite general dialect of CPS in which λs can bind and calls can pass multiple continuations, and continuation references can be captured in closure environments just as value references can [11]. Despite this generality, the uniformity of CPS allows the m-CFA formalism to be given in terms of only a single rule.

Although CPS represents all control transfers as calls, CPS compilers do not typically interpret them naïvely; instead, they recognize the role of continuations in execution and keep them distinct from other values to apply particular compilation strategies, such as allocating continuation closures on the stack [4]. Compilers maintain this distinction by statically partitioning their CPS language into a user world and a continuation world. Terms in the user world correspond to terms in the source program whereas terms in the continuation world are those introduced by the CPS transformation. The distinction is carried into the dynamic semantics as a partition into user- and continuation-world values that respects the static partition: closures over λs are values from the λ's world, are bound exclusively to variables from that world, and are invoked exclusively at call sites from that world.

$$
\begin{array}{rcl@{\qquad}rcl}
pr \in \mathsf{Prgm} & ::= & (\lambda\,(k)\;call) & call \in \mathsf{Call} & = & \mathsf{UCall} + \mathsf{CCall} \\
ucall \in \mathsf{UCall} & ::= & (f\;e\;q)^{l} & ccall \in \mathsf{CCall} & ::= & (q\;e)^{\gamma} \\
f, e \in \mathsf{UExp} & = & \mathsf{UVar} + \mathsf{ULam} & q \in \mathsf{CExp} & = & \mathsf{CVar} + \mathsf{CLam} \\
u \in \mathsf{UVar} & = & \text{a set of identifiers} & k \in \mathsf{CVar} & = & \text{a set of identifiers} \\
ulam \in \mathsf{ULam} & ::= & (\lambda_{l}\,(u\,k)\;call) & clam \in \mathsf{CLam} & ::= & (\lambda_{\gamma}\,(u)\;call) \\
l \in \mathsf{ULab} & = & \text{a set of labels} & \gamma \in \mathsf{CLab} & = & \text{a set of labels}
\end{array}
$$

Fig. 2. A restricted CPS language

Figure 2 presents the grammar of a partitioned CPS language. A call comes from either the user or the continuation world. A call in the user world has operator, value, and continuation arguments; a call in the continuation world has only continuation and value arguments. Arguments are user or continuation expressions which consist of references and λs from the corresponding world. A user-world λ has parameters for its value and continuation, and its body consists of any kind of call. A continuation-world λ body is also any kind of call, but has a parameter only for its value. Each call and λ is annotated with a label specific to its world which distinguishes otherwise-identical terms. A program is a closed λ binding a single continuation which is α-converted, i.e., in which every binding instance of a variable is unique.

After converting the example program to this CPS language, we obtain the program to the right. The entire program becomes a λ awaiting a top-level continuation. The first `let*` binding becomes an immediate application of a `let`-continuation binding `id`. The two subsequent bindings become the corresponding calls to `id` whose continuations bind `y` and `z`, respectively. The body of the `let*` becomes a call to a continuation-aware definition of + which is passed the top-level continuation. User-world labels are drawn from $\{A, B, C, \dots\}$ and continuation-world labels are drawn from $\{a, b, c, \dots\}$.

```
(λ (k0)
  ((λa (id)
     (id 10 (λb (y)
              (id 12 (λc (z)
                       (+ y z k0)^B))^C))^D)
   (λA (x k1) (k1 x)^d))^e)
```

This language is restricted relative to the expressive dialect of CPS that m-CFA supports in two ways: (1) calls pass exactly one continuation and (2) continuation references cannot appear free in the user-world λ which encloses them. These restrictions ensure that expressed programs exhibit only the simple push–pop stack behavior of function calls, in contrast to that of control constructs such as `call/cc` which goes far beyond. Vardoulakis and Shivers [21] present a variant of partitioned CPS they call *restricted CPS* or *RCPS* which imposes the latter restriction but not the former; we call our doubly-restricted variant *R2CPS*.

In R2CPS, the role of a CPS term in the source program can be determined merely from its shape. For example, a tail call in the source program is translated

to a user-world call whose continuation argument is a reference whereas a proper call is translated to one whose continuation argument is a λ. We rely on this ability heavily in the sequel, beginning in the next section.

4 m-CFA^{cps}

R2CPS is a sublanguage of m-CFA's more-general CPS dialect, so a definition of m-CFA over it is no different than m-CFA itself. However, R2CPS allows us to distinguish terms according to the role they play in the source program and specialize the state transition to each. Figure 3 presents m-CFA^{cps}, an R2CPS-restricted m-CFA whose state transition has been factored across (and specialized to) user/continuation and tail/non-tail calls. Because the shape of the con-

$$\Rightarrow_{\hat{\Sigma}_{cps}} \subseteq \hat{\Sigma}_{cps} \times \hat{\Sigma}_{cps}$$

$$((f\ e\ clam)^l, \hat{\rho}, \hat{\sigma}_{cps}) \Rightarrow_{\hat{\Sigma}_{cps}} (call, \hat{\rho}', \hat{\sigma}'_{cps}), \text{ where}$$

$$\begin{array}{rr} ((\lambda_l\,(u\,k)\ call), \hat{\rho}') \in \hat{\mathcal{E}}_{cps}(f, \hat{\rho}, \hat{\sigma}_{cps}) & \hat{d} = \hat{\mathcal{E}}_{cps}(e, \hat{\rho}, \hat{\sigma}_{cps}) \\ \{x_1, \ldots, x_n\} = free((\lambda_l\,(u\,k)\ call)) & \hat{q} = \{(clam, \hat{\rho})\} \\ \hat{\rho}'' = \lfloor l :: \hat{\rho} \rfloor_m & \hat{d}_i = \hat{\sigma}_{cps}(x_i, \hat{\rho}') \end{array}$$

$$\hat{\sigma}'_{cps} = \hat{\sigma}_{cps} \sqcup [(u, \hat{\rho}'') \mapsto \hat{d}] \sqcup [(k, \hat{\rho}'') \mapsto \hat{q}] \sqcup [(x_i, \hat{\rho}'') \mapsto \hat{d}_i]$$

$$((k\ e)^\gamma, \hat{\rho}, \hat{\sigma}_{cps}) \Rightarrow_{\hat{\Sigma}_{cps}} (call, \hat{\rho}', \hat{\sigma}'_{cps}), \text{ where}$$

$$((\lambda_\gamma\,(u)\ call), \hat{\rho}') \in \hat{\sigma}_{cps}(k, \hat{\rho}) \quad \hat{d} = \hat{\mathcal{E}}_{cps}(e, \hat{\rho}, \hat{\sigma}_{cps}) \quad \hat{\sigma}'_{cps} = \hat{\sigma}_{cps} \sqcup [(u, \hat{\rho}') \mapsto \hat{d}]$$

$$((f\ e\ k)^l, \hat{\rho}, \hat{\sigma}_{cps}) \Rightarrow_{\hat{\Sigma}_{cps}} (call, \hat{\rho}', \hat{\sigma}'_{cps}), \text{ where}$$

$$\begin{array}{rr} ((\lambda_l\,(u\,k)\ call), \hat{\rho}') \in \hat{\mathcal{E}}_{cps}(f, \hat{\rho}, \hat{\sigma}_{cps}) & \hat{d} = \hat{\mathcal{E}}_{cps}(e, \hat{\rho}, \hat{\sigma}_{cps}) \\ \{x_1, \ldots, x_n\} = free((\lambda_l\,(u\,k)\ call)) & \hat{q} = \hat{\sigma}_{cps}(k, \hat{\rho}) \\ \hat{\rho}'' = \lfloor l :: \hat{\rho} \rfloor_m & \hat{d}_i = \hat{\sigma}_{cps}(x_i, \hat{\rho}') \end{array}$$

$$\hat{\sigma}'_{cps} = \hat{\sigma}_{cps} \sqcup [(u, \hat{\rho}'') \mapsto \hat{d}] \sqcup [(k, \hat{\rho}'') \mapsto \hat{q}] \sqcup [(x_i, \hat{\rho}'') \mapsto \hat{d}_i]$$

$$(((\lambda_\gamma\,(u)\ call)\ e)^\gamma, \hat{\rho}, \hat{\sigma}_{cps}) \Rightarrow_{\hat{\Sigma}_{cps}} (call', \hat{\rho}', \hat{\sigma}'_{cps}), \text{ where}$$

$$((\lambda_\gamma\,(u')\ call'), \hat{\rho}') \in \{((\lambda_\gamma\,(u)\ call), \hat{\rho})\} \quad \hat{d} = \hat{\mathcal{E}}_{cps}(e, \hat{\rho}, \hat{\sigma}_{cps}) \quad \hat{\sigma}'_{cps} = \hat{\sigma}_{cps} \sqcup [(u', \hat{\rho}) \mapsto \hat{d}]$$

$$\hat{\mathcal{E}}_{cps} : \mathsf{UExp} \times \widehat{Env} \times \widehat{Store}_{cps}$$

$$\hat{\mathcal{E}}_{cps}(u, \hat{\rho}, \hat{\sigma}_{cps}) = \hat{\sigma}_{cps}(u, \hat{\rho}) \qquad \hat{\mathcal{E}}_{cps}(ulam, \hat{\rho}, \hat{\sigma}_{cps}) = \{(ulam, \hat{\rho})\}$$

Fig. 3. R2CPS-restricted m-CFA factored across user/continuation and tail/non-tail calls

tinuation is known, we inline the use of $\hat{\mathcal{E}}_{cps}$ away in each rule. Similarly, because the source world of the operator is known, we inline the use of $\widehat{new}$—which computes the destination environment—away as well. A call $(f\ e\ clam)^l$ corresponds to a non-tail call to the pre-CPS version of f in the source program. A call $(k\ e)^\gamma$ corresponds to a return in the source program. A call $(f\ e\ k)^l$ corresponds to a tail call to the pre-CPS version of f in the source program. Finally, a call $((\lambda_\gamma\,(u)\ call)\ e)^\gamma$ corresponds to a let in the source program.

These rules are merely the sole m-CFA transition rule, limited to R2CPS terms, factored by and specialized to the shape of the call. We capture this fact in the following lemma.

Lemma 1. *For all* $call, call' \in \mathsf{Call}$, $\hat{\rho}, \hat{\rho}' \in \widehat{Env}$, $\hat{\sigma}_{cps}, \hat{\sigma}'_{cps} \in \widehat{Store}$,

$$(call, \hat{\rho}, \hat{\sigma}_{cps}) \Rightarrow_{\tilde{\Sigma}} (call', \hat{\rho}', \hat{\sigma}'_{cps}) \text{ if and only if } (call, \hat{\rho}, \hat{\sigma}_{cps}) \Rightarrow_{\tilde{\Sigma}_{cps}} (call', \hat{\rho}', \hat{\sigma}'_{cps}).$$

A 1CFA^{cps} analysis of the CPS'd example program yields $(\tilde{R}, \hat{\sigma})$ where

$$\tilde{R} = \{(e, \langle\rangle), (D, \langle\rangle), (d, \langle D\rangle), (C, \langle\rangle), (d, \langle C\rangle), (B, \langle\rangle), (+, \langle B\rangle)\}$$

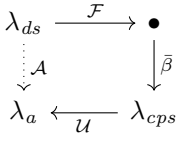
and

$$\begin{array}{llllllllll}
\hat{\sigma} = [(\mathtt{k}_0, \langle\rangle) & \mapsto & \{\mathsf{halt}\}, & (\mathtt{id}, \langle\rangle) & \mapsto & \{(A, \langle\rangle)\}, & (\mathtt{x}, \langle D\rangle) & \mapsto & \{10\}, \\
\quad (\mathtt{k}_1, \langle D\rangle) & \mapsto & \{(b, \langle\rangle)\}, & (\mathtt{y}, \langle\rangle) & \mapsto & \{10\}, & (\mathtt{x}, \langle C\rangle) & \mapsto & \{12\}, \\
\quad (\mathtt{k}_1, \langle C\rangle) & \mapsto & \{(c, \langle\rangle)\}, & (\mathtt{z}, \langle\rangle) & \mapsto & \{12\}, & & & \\
\quad (+_0, \langle B\rangle) & \mapsto & \{10\}, & (+_1, \langle B\rangle) & \mapsto & \{12\}] & & &
\end{array}$$

in which each call is represented by its label. Note that the variables $+_0$ and $+_1$, which correspond to the internal variables of the primitive $+$, are ultimately bound to single, precise values. This precision is an artifact of call-site sensitivity combined with precise call–return correspondence.

5 $\mathcal{A}$-Normal Form

Many compilers [17, 9, 1, 16, 8] convert source programs to CPS in the middle end to do analysis and transformation before generating code. This pipeline is depicted in the diagram to the right where a CPS translation carried out by Fischer's $\mathcal{F}$ [3] operates on a source (direct-style) program in λ_{ds}. However, the CPS translator $\mathcal{F}$ introduces many *administrative redexes* which abstract the continuation within a term. The $\bar{\beta}$ rule reduces these so that repeated application by $\bar{\beta}$ to a normal form results in a term in λ_{cps}. (For our purposes, we can consider λ_{cps} to be R2CPS.)



λ_{cps} terms can be evaluated with a CE machine [4], a machine which manipulates **c**ontrol and **e**nvironment registers—m-CFA's abstract machine is a CE machine augmented with a store register. However, a CE machine models a naïve evaluator which directly interprets CPS terms, allowing the program

itself to manage the continuation. In practice, compilers track the continuation by statically-partitioning the language (as in R2CPS) and manage it directly using a CE machine augmented with a **k**ontinuation register—a CEK machine [2]. This machine uses the shape of each call to determine its role in evaluation. In the call $(k\ e)^\gamma$, for example, the CEK does not look up k in the environment, as a CE machine would do, but instead recognizes this call as a function return and manipulates the continuation register accordingly.

By intercepting the program's continuation management, Flanagan *et al.* [4] observe:

1. Explicit continuation references are unnecessary; only the role of the call matters.
2. The CEK machine effectively inverts the CPS transformation (accurately modeling a code generator).

From these observations they respond in two ways.

First, Flanagan *et al.* devise a set of axioms $\mathcal{A}$ which carry out the corresponding reductions on a λ_{ds} term as $\bar{\beta}$ carries out on a CPS term, thus allowing a λ_a term to be obtained without a round trip through CPS. Reduction by the axioms $\mathcal{A}$ is normalizing, and a term in $\mathcal{A}$-normal form (or ANF) is in λ_a, defined below.

$$\begin{array}{llll} e \in \mathsf{Exp} & ::= \mathsf{let}_\gamma\, x := ce\ \mathsf{in}\, e \mid ce & ce \in \mathsf{CExp} & ::= (ae_0\ ae_1)^l \mid ae^\gamma \\ ae \in \mathsf{AExp} & ::= \lambda_l x.e \mid x & x \in \mathsf{Var} & = \text{a set of identifiers} \end{array}$$

Programs in λ_a lack explicit continuations but, like CPS, name all intermediate values. A *proper expression* e is a let expression, which binds a call expression ce to a variable whose scope is another proper expression, or a call expression itself. A *call expression* ce is an atomic expression ae^γ or an application $(ae_0\ ae_1)^l$. An *atomic expression* ae is a variable reference x or a λ abstraction $\lambda_l x.e$. A program in λ_a is a closed expression that is α-converted. Call expressions are annotated with the user-world labels of λ_{cps}; let expressions and atomic expressions are annotated with continuation-world labels. The set of λs $\lambda_l x.e$ is Lam.

Second, Flanagan *et al.* define a function $\mathcal{U}$ that strips CPS terms of redundant continuation information, converting λ_{cps} terms to λ_a terms. We present $\mathcal{U}$ in Figure 4 as well as its inverse $\mathcal{U}^{-1}$. Defining $\mathcal{U}^{-1}$ is less straightforward than defining $\mathcal{U}$ because $\mathcal{U}$ removes continuation references but $\mathcal{U}^{-1}$ must synthesize them. To make synthesis straightforward, we define the set λ_{cps}^{WN} of well-named R2CPS programs. A R2CPS program pr is *well-named* if, for each user-world function $(\lambda_l\,(u\,k)\,call)$, the name of k is derivable from $\mathcal{U}_e[\![call]\!]$ by $\mathcal{U}_k : \mathsf{Exp} \to \mathsf{CVar}$ and vice versa by $\mathcal{U}_k^{-1} : \mathsf{CVar} \to \mathsf{Exp}$. This correspondence between an ANF expression and a continuation-world variable helps us build a correspondence between different formulations of m-CFA (cf. §7). Any R2CPS program can be α-converted to one that is well-named, so λ_{cps}^{WN} is not materially smaller than λ_{cps}. The $\mathcal{U}$ and $\mathcal{U}^{-1}$ definitions are supported by variable conversion functions $\mathcal{U}_x : \mathit{UVar} \to \mathit{Var}$ and $\mathcal{U}_x^{-1}\,\mathit{Var} \to \mathit{UVar}$ which convert between

$$\mathcal{U} : \lambda_{cps}^{WN} \to \lambda_a \qquad \mathcal{U}^{-1} : \lambda_a \to \lambda_{cps}^{WN}$$

$$\mathcal{U}[\![(\lambda\,(k)\;call)]\!] = \mathcal{U}_e[\![call]\!] \qquad \mathcal{U}^{-1}[\![pr]\!] = (\lambda\,(k)\,\mathcal{U}_e^{-1}[\![pr]\!](k)) \text{ where } k = \mathcal{U}_k[\![pr]\!]$$

$$\mathcal{U}_e : \mathsf{Call} \to \mathsf{Exp} \qquad \mathcal{U}_e^{-1} : \mathsf{Exp} \to \mathsf{CVar} \to \mathsf{Call}$$

$$\mathcal{U}_e[\![((\lambda_{\gamma'}\;(u)\;call)\;e)^{\gamma}]\!] = \mathsf{let}_{\gamma'}\,\mathcal{U}_x[\![u]\!] := \mathcal{U}_{ae}[\![e]\!]^{\gamma}\,\mathsf{in}\,\mathcal{U}_e[\![call]\!]$$

$$\mathcal{U}_e[\![(f\;e\;(\lambda_{\gamma}\;(u)\;call))^{l}]\!] = \mathsf{let}_{\gamma}\,\mathcal{U}_x[\![u]\!] := (\mathcal{U}_{ae}[\![f]\!]\;\mathcal{U}_{ae}[\![e]\!])^{l}\,\mathsf{in}\,\mathcal{U}_e[\![call]\!]$$

$$\mathcal{U}_e[\![(k\;e)^{\gamma}]\!] = \mathcal{U}_{ae}[\![e]\!]^{\gamma}$$

$$\mathcal{U}_e[\![(f\;e\;k)^{l}]\!] = (\mathcal{U}_{ae}[\![f]\!]\;\mathcal{U}_{ae}[\![e]\!])^{l}$$

$$\mathcal{U}_e^{-1}[\![\mathsf{let}_{\gamma}\,x := (ae_0\;\;ae_1)^{l}\,\mathsf{in}\,e]\!](k) = (\mathcal{U}_{ae}^{-1}[\![ae_0]\!]\;\mathcal{U}_{ae}^{-1}[\![ae_1]\!]\;(\lambda_{\gamma}\,(\mathcal{U}_x^{-1}[\![x]\!])\,\mathcal{U}_e^{-1}[\![e]\!](k)))^{l}$$

$$\mathcal{U}_e^{-1}[\![\mathsf{let}_{\gamma}\,x := ae^{\gamma'}\,\mathsf{in}\,e]\!](k) = ((\lambda_{\gamma}\,(\mathcal{U}_x^{-1}[\![x]\!])\,\mathcal{U}_e^{-1}[\![e]\!](k))\;\mathcal{U}_{ae}^{-1}[\![ae]\!])^{\gamma'}$$

$$\mathcal{U}_e^{-1}[\![(ae_0\;\;ae_1)^{l}]\!](k) = (\mathcal{U}_{ae}^{-1}[\![ae_0]\!]\;\mathcal{U}_{ae}^{-1}[\![ae_1]\!]\;\;k)^{l}$$

$$\mathcal{U}_e^{-1}[\![ae^{\gamma}]\!](k) = (k\;\mathcal{U}_{ae}^{-1}[\![ae]\!])^{\gamma}$$

$$\mathcal{U}_{ae} : \mathsf{UExp} \to \mathsf{AExp} \qquad \mathcal{U}^{-1} : \mathsf{AExp} \to \mathsf{UExp}$$

$$\mathcal{U}[\![(\lambda_l\,(u\,k)\;call)]\!] = \lambda_l \mathcal{U}_x[\![u]\!].\mathcal{U}_e[\![call]\!] \qquad \mathcal{U}^{-1}[\![\lambda_l x.e]\!] = (\lambda_l\,(\mathcal{U}_x^{-1}[\![x]\!]\,k)\,\mathcal{U}_e^{-1}[\![e]\!](k)) \text{ where } k = \mathcal{U}_k[\![e]\!]$$

$$\mathcal{U}[\![u]\!] = \mathcal{U}_x[\![u]\!] \qquad \mathcal{U}^{-1}[\![x]\!] = \mathcal{U}_x^{-1}[\![x]\!]$$

Fig. 4. The λ_{cps}^{WN}–λ_a bijection pair $\mathcal{U}/\mathcal{U}^{-1}$.

user-world variables in λ_{cps} and variables in λ_a. These functions precisely preserve user- and continuation-world labels. The following lemma establishes that these functions are indeed mutual inverses.

Lemma 2. *$\mathcal{U}^{-1} \circ \mathcal{U} = I_{\lambda_{cps}^{WN}}$ and $\mathcal{U} \circ \mathcal{U}^{-1} = I_{\lambda_a}$*

Using $\mathcal{U}$ to convert the CPS version of the example program yields the program to the right. While user-world labels remain associated with their corresponding λ or call, continuation-world labels on λs annotate lets and on calls annotate atomic expressions.

$$\begin{array}{l}\mathsf{let}_a\;\mathtt{id} = (\lambda_A \mathtt{x}.\mathtt{x}^{d})^{e}\\ \mathsf{in}\,\mathsf{let}_b\;\mathtt{y} = (\mathtt{id}\;10)^{D}\\ \quad\mathsf{in}\,\mathsf{let}_c\;\mathtt{z} = (\mathtt{id}\;12)^{C}\\ \qquad\mathsf{in}\,(+\;\mathtt{y}\;\mathtt{z})^{B}\end{array}$$

6 m-CFA^a

We now define m-CFA^a, m-CFA for λ_a. We then extend the term isomorphism of §5 to show that an m-CFA^a analysis is isomorphic to a m-CFA^{cps} analysis, thus demonstrating that the continuation references in λ_{cps} are redundant with respect to m-CFA just as they are for a CEK machine.

m-CFA^a is defined in terms of an abstract CEK machine using the *Abstracting Abstracting Machines* (AAM) methodology [19]. Whereas the continuation register contains a representation of continuation, such as a stack, in a concrete CEK machine, it contains the address of a store-allocated continuation in an abstract CEK machine. (Anticipating our application of P4F in §8, we separate values and continuations into dedicated stores.)

Continuation variables, used to form continuation addresses in m-CFA^{cps}, are not present in m-CFA^a. However, we can use $\mathcal{U}_k$ correspondence of each λ_{cps} continuation variable to the λ_a representation of its scope to obtain a m-CFA^a continuation address from each corresponding one in m-CFA^{cps}. Thus, a m-CFA^a continuation address consists of an expression entailing a continuation scope—a λ body or the program itself—paired with an environment. For a fixed program pr (with unique labels), the body of the innermost-enclosing λ of any expression is apparent; we assume a function $\zeta_{pr} : \mathit{Exp} \to \mathit{Exp}$ which produces the body of the innermost-enclosing λ of the given expression, or the entire program if it is not enclosed. This function allows us to derive the continuation address $(\zeta_{pr}(e), \hat{\rho})$ from the CE registers $(e, \hat{\rho})$ and in turn omit the K register from configurations altogether.

Figure 5 presents m-CFA^a system space and transfer function. Evaluation of the let-binding of a call creates an abstract frame $\mathsf{ar}(x, e, \hat{\rho})$ which consists of the bound variable, body expression, and environment. This frame contains a link to the previous frame, but only implicitly, as we will see momentarily. The transition constructs an environment for the call and extends the value store, copying bindings of free variables, in the standard way. For the continuation address, it uses the body expression of the called procedure paired with its environment, in correspondence to the continuation variable of its CPS representation. Evaluation of an atomic expression, which represents a function return, looks up the top frame to bind the return value and continue evaluation. The continuation address is derived from the atomic expression itself using ζ_{pr}. The atomic expression's value is bound in the store and the expression and environment within the stack frame are restored. Evaluation of a tail call is precisely the same as for a let-bound call, except that the current continuation is obtained by synthesizing the continuation address using ζ_{pr} and copied to the callee's continuation address. Similarly, evaluation of a let-bound atomic expression proceeds precisely the same as for an atomic expression, except that the continuation to which the value is "returned" is local.

An m-CFA^a analysis is defined as the least fixed point of the function $\Rightarrow_{\tilde{\Xi}_a}$, which is computed in the same way as $\Rightarrow_{\tilde{\Xi}_{cps}}$.

An $[m = 1]$-CFA^a analysis of the ANF'd example program yields $(\tilde{R}, \hat{\sigma}, \tilde{\sigma}_\kappa)$ where

$$\tilde{R} = \{(e, \langle\rangle), (D, \langle\rangle), (d, \langle D\rangle), (C, \langle\rangle), (d, \langle C\rangle), (B, \langle\rangle), (+, \langle B\rangle)\}$$

and

$$\begin{array}{rcl@{\quad}rcl@{\quad}rcl}
\hat{\sigma} = [(\mathtt{id}, \langle\rangle) & \mapsto & \{(A, \langle\rangle)\}, & (\mathtt{x}, \langle D\rangle) & \mapsto & \{10\}, & (\mathtt{y}, \langle\rangle) & \mapsto & \{10\}, \\
(\mathtt{x}, \langle C\rangle) & \mapsto & \{12\}, & (\mathtt{z}, \langle\rangle) & \mapsto & \{12\}, & (+_0, \langle B\rangle) & \mapsto & \{10\}, \\
(+_1, \langle B\rangle) & \mapsto & \{12\}] & & & & & &
\end{array}$$

and

$$\begin{array}{rcl@{\quad}rcl}
\tilde{\sigma}_\kappa = [(\mathtt{k}_0, \langle\rangle) & \mapsto & \{\mathtt{mt}\}, & (d, \langle D\rangle) & \mapsto & \{(\mathtt{y}, b, \langle\rangle)\}, \\
(d, \langle C\rangle) & \mapsto & \{(\mathtt{z}, c, \langle\rangle)\}]. & & &
\end{array}$$

$$\tilde{\varsigma} \in \tilde{\Sigma}_a = \mathsf{Exp} \times \widehat{Env} \times \widehat{Store}_a \times \widetilde{KStore}$$

$$\begin{array}{lll}
\hat{\sigma}_a \in \widehat{Store}_a & = \widehat{Addr}_a \to \widehat{D}_a & \widehat{Addr}_a = \mathsf{Var} \times \widehat{Env} \\
\tilde{\sigma}_\kappa \in \widetilde{KStore} & = \widetilde{KAddr} \to \mathcal{P}(\widetilde{Frame}) & \widetilde{KAddr} = \mathsf{Exp} \times \widehat{Env} \\
\hat{d}^a \in \widehat{D}_a & = \mathcal{P}(\mathsf{Lam} \times \widehat{Env}) & \widetilde{Frame}_a ::= \mathsf{mt} \mid \mathsf{ar}(x, e, \hat{\rho})
\end{array}$$

$$\Rightarrow_{\tilde{\Sigma}_a} \subseteq \tilde{\Sigma}_a \times \tilde{\Sigma}_a$$

$$(\mathsf{let}_\gamma\ x := (ae_0\ ae_1)^l\ \mathsf{in}\ e, \hat{\rho}, \hat{\sigma}_a, \tilde{\sigma}_\kappa) \Rightarrow_{\tilde{\Sigma}_a} (e', \hat{\rho}'', \hat{\sigma}'_a, \tilde{\sigma}'_\kappa), \text{ where}$$

$$\begin{array}{ll}
(\lambda_l x'.e', \hat{\rho}') \in \hat{\mathcal{E}}_a(ae_0, \hat{\rho}, \hat{\sigma}_a) & \hat{d}^a = \hat{\mathcal{E}}_a(ae_1, \hat{\rho}, \hat{\sigma}_a) \\
\{x_1, \ldots, x_n\} = \mathit{free}(\lambda_l x.e') & \tilde{\phi} = \{\mathsf{ar}(x, e, \hat{\rho})\} \\
\hat{\rho}'' = \lfloor l :: \hat{\rho} \rfloor_m & \hat{d}^a_i = \hat{\sigma}_a(x_i, \hat{\rho}') \\
\multicolumn{2}{c}{\hat{\sigma}'_a = \hat{\sigma}_a \sqcup [(x, \hat{\rho}'') \mapsto \hat{d}^a] \sqcup [(x_i, \hat{\rho}'') \mapsto \hat{d}^a_i]} \\
\multicolumn{2}{c}{\tilde{\sigma}'_\kappa = \tilde{\sigma}_\kappa \sqcup [(e', \hat{\rho}'') \mapsto \tilde{\phi}]}
\end{array}$$

$$(ae^\gamma, \hat{\rho}, \hat{\sigma}_a, \tilde{\sigma}_\kappa) \Rightarrow_{\tilde{\Sigma}_a} (e, \hat{\rho}', \hat{\sigma}'_a, \tilde{\sigma}_\kappa), \text{ where}$$

$$\mathsf{ar}(x, e, \hat{\rho}') \in \tilde{\sigma}_\kappa(\zeta_{pr}[\![ae^\gamma]\!], \hat{\rho}) \qquad \hat{d}^a = \hat{\mathcal{E}}_a(ae, \hat{\rho}, \hat{\sigma}_a) \qquad \hat{\sigma}'_a = \hat{\sigma}_a \sqcup [(x, \hat{\rho}') \mapsto \hat{d}^a]$$

$$((ae_0\ ae_1)^l, \hat{\rho}, \hat{\sigma}_a, \tilde{\sigma}_\kappa) \Rightarrow_{\tilde{\Sigma}_a} (e', \hat{\rho}'', \hat{\sigma}'_a, \tilde{\sigma}'_\kappa), \text{ where}$$

$$\begin{array}{ll}
(\lambda_l x.e', \hat{\rho}') \in \hat{\mathcal{E}}_a(ae_0, \hat{\rho}, \hat{\sigma}_a) & \hat{d}^a = \hat{\mathcal{E}}_a(ae_1, \hat{\rho}, \hat{\sigma}_a) \\
\{x_1, \ldots, x_n\} = \mathit{free}(\lambda_l x.e') & \tilde{\phi} = \tilde{\sigma}_\kappa(\zeta_{pr}[\![(ae_0\ ae_1)^l]\!], \hat{\rho}) \\
\hat{\rho}'' = \lfloor l :: \hat{\rho} \rfloor_m & \hat{d}^a_i = \hat{\sigma}_a(x_i, \hat{\rho}') \\
\multicolumn{2}{c}{\hat{\sigma}'_a = \hat{\sigma}_a \sqcup [(x, \hat{\rho}'') \mapsto \hat{d}^a] \sqcup [(x_i, \hat{\rho}'') \mapsto \hat{d}^a_i]} \\
\multicolumn{2}{c}{\tilde{\sigma}'_\kappa = \tilde{\sigma}_\kappa \sqcup [(e', \hat{\rho}'') \mapsto \tilde{\phi}]}
\end{array}$$

$$(\mathsf{let}_\gamma\ x := ae^\gamma\ \mathsf{in}\ e, \hat{\rho}, \hat{\sigma}_a, \tilde{\sigma}_\kappa) \Rightarrow_{\tilde{\Sigma}_a} (e', \hat{\rho}', \hat{\sigma}'_a, \tilde{\sigma}_\kappa), \text{ where}$$

$$\mathsf{ar}(x', e', \hat{\rho}') \in \{\mathsf{ar}(x, e, \hat{\rho})\} \qquad \hat{d}^a = \hat{\mathcal{E}}_a(ae, \hat{\rho}, \hat{\sigma}_a) \qquad \hat{\sigma}'_a = \hat{\sigma}_a \sqcup [(x', \hat{\rho}') \mapsto \hat{d}^a]$$

$$\hat{\mathcal{E}}_a : \mathsf{AExp} \times \widehat{Env} \times \widehat{Store}_a$$

$$\hat{\mathcal{E}}_a(x, \hat{\rho}, \hat{\sigma}_a) = \hat{\sigma}_a(x, \hat{\rho}) \qquad \hat{\mathcal{E}}_a(\lambda_l x.e, \hat{\rho}, \hat{\sigma}_a) = \{(\lambda_l x.e, \hat{\rho})\}$$

$$\tilde{\xi}_a \in \tilde{\Xi}_a = \tilde{R}_a \times \widehat{Store}_a \times \widetilde{KStore} \qquad \tilde{r}_a \in \tilde{R}_a = \mathcal{P}(\tilde{C}_a) \qquad \tilde{c}_a \in \tilde{C}_a = \mathsf{Exp} \times \widehat{Env}$$

$$\Rightarrow_{\tilde{\Xi}_a} : \tilde{\Xi}_a \to \tilde{\Xi}_a$$

$$(\tilde{C}_a, \hat{\sigma}_a, \tilde{\sigma}_\kappa) \Rightarrow_{\tilde{\Xi}_a} (\tilde{C}_a^{init} \cup \tilde{C} \cup \tilde{C}', \hat{\sigma}'_a, \tilde{\sigma}_\kappa^{init} \sqcup \tilde{\sigma}'_\kappa) \text{ where}$$

$$\begin{array}{ll}
\tilde{C}_a^{init} = \{(pr, \langle\rangle)\} & \tilde{\sigma}_\kappa^{init} = [(pr, \langle\rangle) \mapsto \{\mathsf{mt}\}] \\
\tilde{S}'_a = \{\tilde{\varsigma}'_a : \tilde{c}_a \in \tilde{C}_a \text{ and } (\tilde{c}_a, \hat{\sigma}_a, \tilde{\sigma}_\kappa) \Rightarrow_{\tilde{\Sigma}_a} \tilde{\varsigma}'_a\} & \hat{\sigma}'_a = \bigsqcup_{(\tilde{c}_a, \hat{\sigma}_a, \tilde{\sigma}_\kappa) \in \tilde{S}'_a} \hat{\sigma}_a \\
\tilde{C}'_a = \{\tilde{c}_a : (\tilde{c}_a, \hat{\sigma}_a, \tilde{\sigma}_\kappa) \in \tilde{S}'_a\} & \tilde{\sigma}'_\kappa = \bigsqcup_{(\tilde{c}_a, \hat{\sigma}_a, \tilde{\sigma}_\kappa) \in \tilde{S}'_a} \tilde{\sigma}_\kappa
\end{array}$$

Fig. 5. m-CFA^a

Value store allocations are identical to user-world allocations in m-CFA^{cps}. Continuation frames (in which an expression is represented by its label) correspond directly to continuation-world allocations, as we show in the next section.

7 m-CFA^{cps}–m-CFA^{a} Correspondence

We extend the λ^{WN}_{cps}–λ_a isomorphism through $\mathcal{U}$ to m-CFA^{cps}–m-CFA^a, first to the state space, then to transition rules, and then finally to the entire analysis. The definitions

$$\mathcal{U}(call, \hat{\rho}, \hat{\sigma}_{cps}) = (\mathcal{U}_e[\![call]\!], \hat{\rho}, \hat{\sigma}_a, \tilde{\sigma}_\kappa) \text{ where } (\hat{\sigma}_a, \tilde{\sigma}_\kappa) = T(\hat{\sigma}_{cps})$$

and

$$\mathcal{U}^{-1}(e, \hat{\rho}, \hat{\sigma}_a, \tilde{\sigma}_\kappa) = (\mathcal{U}_e^{-1}[\![e]\!](\mathcal{U}_k^{-1}(\zeta_{pr}[\![e]\!])), T^{-1}(\hat{\sigma}_a, \tilde{\sigma}_\kappa))$$

extend it to the state space. T_{pr}/T_{pr}^{-1}, seen in Figure 6, is a lattice isomorphism (i.e. it is a bijection which commutes with the join operation and refinement relation) between the m-CFA^{cps} store lattice and the m-CFA^a value and continuation store product lattice. The T_{pr}/T_{pr}^{-1} isomorphism induces an isomorphism

$$T_{pr} : \widehat{Store}_{cps} \to \widehat{Store}_a \times \widetilde{KStore}$$

$$T_{pr}(\hat{\sigma}_{cps}) = (\hat{\sigma}_a, \tilde{\sigma}_\kappa) \text{ where}$$

$$\begin{aligned}
\hat{\sigma}_a &= \lambda(x, \hat{\rho}).\{(\lambda_l \mathcal{U}_x[\![u]\!].\mathcal{U}_e[\![call]\!], \hat{\rho}') : ((\lambda_l\,(u\,k)\;call), \hat{\rho}') \in \hat{\sigma}_{cps}(\mathcal{U}_x^{-1}[\![x]\!], \hat{\rho})\} \\
\tilde{\sigma}_\kappa &= \lambda(e, \hat{\rho}).\{\mathsf{ar}(\mathcal{U}_x[\![u]\!], \mathcal{U}_e[\![call]\!], \hat{\rho}') : ((\lambda_\gamma\,(u)\;call), \hat{\rho}') \in \hat{\sigma}_{cps}(\mathcal{U}_k^{-1}(\zeta_{pr}[\![e]\!]), \hat{\rho})\} \\
&\quad \cup \{\mathsf{mt} : \mathsf{halt} \in \hat{\sigma}_{cps}(\mathcal{U}_k^{-1}(\zeta_{pr}[\![e]\!]), \hat{\rho})\}
\end{aligned}$$

$$T_{pr}^{-1} : \widehat{Store}_a \times \widetilde{KStore} \to \widehat{Store}_{cps}$$

$$T_{pr}^{-1}(\hat{\sigma}_a, \tilde{\sigma}_\kappa) = \lambda(z, \hat{\rho}).\begin{cases} \{((\lambda_l\,(\mathcal{U}_x^{-1}[\![x]\!]\,k)\,\mathcal{U}_e^{-1}[\![e]\!](k)), \hat{\rho}') : (\lambda_l x.e, \hat{\rho}') \in \sigma(z, \hat{\rho})\}, \text{ if } z = (u, 0) \\ \quad \{((\lambda_\gamma\,(\mathcal{U}_x^{-1}[\![x]\!])\,\mathcal{U}_e^{-1}[\![e]\!](k)), \hat{\rho}') : \mathsf{ar}(x, e, \hat{\rho}') \in \tilde{\sigma}_\kappa(\zeta_{pr}[\![\mathcal{U}_k^{-1}(k)]\!], \hat{\rho})\} \\ \cup \{\mathsf{halt} : \mathsf{mt} \in \tilde{\sigma}_\kappa(\zeta_{pr}[\![\mathcal{U}_k^{-1}(k)]\!], \hat{\rho})\}, \text{ if } z = (k, 1) \end{cases}$$

Fig. 6. T_{pr}/T_{pr}^{-1} lattice isomorphism between $\widetilde{Store}_{cps}$ and $\widehat{Store}_a \times \widetilde{KStore}$

between the system spaces $\tilde{\Xi}_{cps}$ and $\tilde{\Xi}_a$ (elided for space). We also use T_{pr}/T_{pr}^{-1} to establish that $\Rightarrow_{\tilde{\Sigma}_{cps}}$ and $\Rightarrow_{\tilde{\Sigma}_a}$ are isomorphic, which is proved straightforwardly since each transition rule in m-CFA^a corresponds to a transition rule in m-CFA^{cps}.

Lemma 3 ($\Rightarrow_{\tilde{\Sigma}_{cps}}$–$\Rightarrow_{\tilde{\Sigma}_a}$ Isomorphism). *For all $\tilde{\varsigma}_{cps}, \tilde{\varsigma}'_{cps} \in \tilde{\Sigma}_{cps}$, $\tilde{\varsigma}_{cps} \Rightarrow_{\tilde{\Sigma}_{cps}} \tilde{\varsigma}'_{cps} \iff \mathcal{U}(\tilde{\varsigma}_{cps}) \Rightarrow_{\tilde{\Sigma}_a} \mathcal{U}(\tilde{\varsigma}'_{cps})$ and, for all $\tilde{\varsigma}, \tilde{\varsigma}' \in \tilde{\Sigma}_a$, $\tilde{\varsigma} \Rightarrow_{\tilde{\Sigma}_a} \tilde{\varsigma}' \iff \mathcal{U}^{-1}(\tilde{\varsigma}_a) \Rightarrow_{\tilde{\Sigma}_{cps}} \mathcal{U}^{-1}(\tilde{\varsigma}'_a)$.*

From the $\tilde{\Xi}_{cps}$–$\tilde{\Xi}_a$ isomorphism and the $\Rightarrow_{\tilde{\Sigma}_{cps}}$–$\Rightarrow_{\tilde{\Sigma}_a}$ isomorphism, we can show that $\Rightarrow_{\tilde{\Xi}_{cps}}$ and $\Rightarrow_{\tilde{\Xi}_a}$ commute with the isomorphism.

Theorem 1 (m-CFA^{cps}–m-CFA^{a} Correspondence). *The following diagram commutes.*

$$\begin{array}{ccc} \tilde{\Xi}_{cps} & \xRightarrow{cps} & \tilde{\Xi}_{cps} \\ {\scriptstyle \mathcal{U}/\mathcal{U}^{-1}}\uparrow\downarrow & & \uparrow\downarrow{\scriptstyle \mathcal{U}/\mathcal{U}^{-1}} \\ \tilde{\Xi}_{a} & \xRightarrow{a} & \tilde{\Xi}_{a} \end{array}$$

An immediate corollary is that the least fixed points of $\Rightarrow_{\tilde{\Xi}_{cps}}$ and $\Rightarrow_{\tilde{\Xi}_{a}}$ correspond to one another so that m-CFA^{cps} and m-CFA^{a} compute the same analysis.

8 Perfect Stack Precision

k-CFA and m-CFA each model the execution of a program using a finite state machine (FSM) in which the nodes are execution states and the edges are control transitions. This model has the benefit that it is easy to construct using a straightforward workset algorithm. However, it is unable to capture the call–return behavior of programs whose execution is mediated by a stack. In particular, it cannot precisely associate returns to points of call, instead discovering spurious control paths within the program's execution.

In the same year that m-CFA was presented, Vardoulakis and Shivers [20] presented CFA2, a *stack-precise* CFA which models program execution with a pushdown system instead of an FSM. Unlike an FSM, a pushdown model allows to analysis to precisely associate each return to its corresponding call, thereby significantly increasing control precision. Unfortunately, CFA2 suffers from the shortcomings that (1) it is context-insensitive (i.e. *monovariant*); (2) its algorithm is in EXPTIME, and (3) its algorithm uses a relatively-complex summarization-based approach.

Follow-on work mitigated each of these shortcomings, achieving context sensitivity, low computational complexity, and algorithmic simplicity [10, 7]. In one fell swoop, Gilray *et al.* [6] resolved them all with the *pushdown for free* (or *P4F*) technique to achieve perfect stack precision "for free" both in the sense that it requires essentially no implementation effort and also in the sense that it doesn't increase the computational complexity of the target CFA. The technique derives from two observations:

1. In an abstract-machine based CFA, the stack precision is determined by the continuation allocator, a (often implicit) function $\widehat{alloc}_\kappa : \hat{\Sigma} \times Exp \times \widehat{Env} \times \widehat{Store} \to \widehat{Addr}_\kappa$ of the source configuration $\hat{\varsigma} \in \hat{\Sigma}$ and the target expression $e \in Exp$, environment $\hat{\rho} \in \widehat{Env}$, and store $\sigma \in \widehat{Store}$.
2. Perfect stack precision is achieved when, within the same abstract invocation, the set of continuations for an exit configuration is no less precise than that of the corresponding entry configuration.

The technique entails only the following continuation allocator, by which an address consists solely of the target expression and environment.

$$\widehat{alloc}_\kappa(\hat{\varsigma}, e, \hat{\rho}, \hat{\sigma}) = (e, \hat{\rho})$$

In essence, the continuation address is the entry configuration itself (when the store is factored out into the system space), which ensures that it only ever refers to a single such configuration.

The P4F technique is formulated in an ANF setting; having formulated m-CFA in such a setting, we are now positioned to apply P4F to realize a stack-precise variant of m-CFA, which we set out to do in the next section.

9 m-CFA is Stack-Precise

The application of P4F to achieve perfect stack precision is straightforward: on a call transition, allocate the continuation at an address consisting of the target configuration's expression and environment. By inspection, it is clear that m-CFA^a *already* uses this allocation strategy and consequently is already stack-precise. It follows from Theorem 1 that R2CPS-limited *m-CFA is and always has been stack-precise.* (We discuss this corollary in §10.)

While our primary result is largely in hand, we review the key pieces of the proof of precision and discuss the modifications needed to account for tail calls, which our setting has but P4F's doesn't.

9.1 Overview of Stack Precision

The property of *precision*—also called *completeness*—is dual to the property of *soundness*. Whereas soundness conveys that every behavior in the reference semantics is present in the abstract semantics, *completeness* conveys that no other behavior is present. With respect to stacks, completeness means that every stack implied by the abstract semantics is realizable by a reference semantics which represents stacks explicitly. We now present this reference semantics $\Rightarrow_{\hat{\Sigma}_a}$, show that the abstract semantics $\Rightarrow_{\tilde{\Sigma}_a}$ are sound with respect to $\Rightarrow_{\hat{\Sigma}_a}$, define what it means for a stack to be realizable by $\Rightarrow_{\hat{\Sigma}_a}$ and implied by $\Rightarrow_{\tilde{\Sigma}_a}$, and finally prove that every stack implied by $\Rightarrow_{\tilde{\Sigma}_a}$ is realizable by $\Rightarrow_{\hat{\Sigma}_a}$. (We elide the straightforward result that $\Rightarrow_{\hat{\Sigma}_a}$ is sound with respect to a concrete reference semantics for space.)

Figure 7 presents a small-step semantics for λ_a in which each configuration includes a stack instead of a continuation store. Except for the handling of the continuation, this semantics is identical to the abstract semantics. When an atomic expression is let-bound or the call is a tail call, the continuation is undisturbed. When a call expression is let-bound, a frame is pushed on the continuation. Evaluation of an atomic expression pops the top frame and restores its expression and environment as it binds the result.

An analysis in the system space $\hat{\Xi}_a$ is defined as the least fixed point of $\Rightarrow_{\hat{\Sigma}_a}$, which is defined similarly to $\Rightarrow_{\tilde{\Sigma}_a}$. However, unlike the abstract system space $\tilde{\Xi}_a$, the system space $\hat{\Xi}_a$ is infinite due to unbounded stacks within configurations. Consequently, the least fixed point of $\Rightarrow_{\hat{\Sigma}_a}$ is well-defined but incomputable.

$$\tilde{\varsigma}_a \in \tilde{\Sigma}_a = \mathsf{Exp} \times \widehat{Env}_a \times \widehat{Store}_a \times \widehat{Stack}_a$$
$$\hat{\kappa}_a \in \widehat{Stack}_a ::= \mathsf{mt} \mid \mathsf{ar}(x, e, \hat{\rho}_a, \hat{\kappa}_a)$$
$$\Rightarrow_{\hat{\Sigma}_a} \subseteq \hat{\Sigma}_a \times \hat{\Sigma}_a$$

$$(\mathsf{let}_\gamma\, x := ae^\gamma \,\mathsf{in}\, e, \hat{\rho}, \hat{\sigma}_a, \hat{\kappa}) \Rightarrow_{\hat{\Sigma}_a} (e, \hat{\rho}, \hat{\sigma}'_a, \hat{\kappa}) \qquad (ae^\gamma, \hat{\rho}, \hat{\sigma}_a, \hat{\kappa}) \Rightarrow_{\hat{\Sigma}_a} (e, \hat{\rho}', \hat{\sigma}'_a, \hat{\kappa}')$$
$$\hat{d}^a = \hat{\mathcal{E}}_a(ae, \hat{\rho}, \hat{\sigma}_a) \quad \hat{\sigma}'_a = \hat{\sigma}_a \sqcup [(x, \hat{\rho}) \mapsto \hat{d}^a] \qquad \hat{d}^a = \hat{\mathcal{E}}_a(ae, \hat{\rho}, \hat{\sigma}_a) \quad \hat{\kappa} = \mathsf{ar}(x, e, \hat{\rho}', \hat{\kappa}')$$
$$\hat{\sigma}'_a = \hat{\sigma}_a \sqcup [(x, \hat{\rho}') \mapsto \hat{d}^a]$$
$$(\mathsf{let}_\gamma\, x := (ae_0\ ae_1)^l \,\mathsf{in}\, e, \hat{\rho}, \hat{\sigma}_a, \hat{\kappa}) \Rightarrow_{\hat{\Sigma}_a} (e', \hat{\rho}'', \hat{\sigma}'_a, \hat{\kappa}')((ae_0\ ae_1)^l, \hat{\rho}, \hat{\sigma}_a, \hat{\kappa}) \Rightarrow_{\hat{\Sigma}_a} (e', \hat{\rho}'', \hat{\sigma}'_a, \hat{\kappa})$$

$$(\lambda_l x'.e', \hat{\rho}') \in \hat{\mathcal{E}}_a(ae_0, \hat{\rho}, \hat{\sigma}_a) \quad \hat{d}^a = \hat{\mathcal{E}}_a(ae_1, \hat{\rho}, \hat{\sigma}_a) \quad (\lambda_l x.e', \hat{\rho}') \in \hat{\mathcal{E}}_a(ae_0, \hat{\rho}, \hat{\sigma}_a) \quad \hat{d}^a = \hat{\mathcal{E}}_a(ae_1, \hat{\rho}, \hat{\sigma}_a)$$
$$\{x_1, \ldots, x_n\} = \mathit{free}(\lambda_l x.e') \quad \hat{\kappa}' = \mathsf{ar}(x, e, \hat{\rho}, \hat{\kappa}) \quad \{x_1, \ldots, x_n\} = \mathit{free}(\lambda_l x.e') \quad \hat{d}^a_i = \hat{\sigma}_a(x_i, \hat{\rho}')$$
$$\hat{\rho}'' = \lfloor l :: \hat{\rho} \rfloor_m \quad \hat{d}^a_i = \hat{\sigma}_a(x_i, \hat{\rho}') \qquad \hat{\rho}'' = \lfloor l :: \hat{\rho} \rfloor_m$$
$$\hat{\sigma}'_a = \hat{\sigma}_a \sqcup [(x, \hat{\rho}'') \mapsto \hat{d}^a] \sqcup [(x_i, \hat{\rho}'') \mapsto \hat{d}^a_i] \qquad \hat{\sigma}'_a = \hat{\sigma}_a \sqcup [(x, \hat{\rho}'') \mapsto \hat{d}^a] \sqcup [(x_i, \hat{\rho}'') \mapsto \hat{d}^a_i]$$

Fig. 7. State transition rules $\Rightarrow_{\hat{\Sigma}_a}$

We relate the abstract state space $\tilde{\Sigma}_a$ and stack state space $\hat{\Sigma}_a$ by way of an abstraction function $|\cdot| : \hat{\Sigma}_a \to \tilde{\Sigma}_a$ where

$$|(e, \hat{\rho}, \hat{\sigma}_a, \hat{\kappa})| = (e, \hat{\rho}, \hat{\sigma}_a, F(\zeta_{pr}[\![e]\!], \hat{\rho}, \hat{\kappa}))$$

The F metafunction allocates a stack frame-by-frame to produce a continuation store in which all frames are allocated. It relies on the ζ_{pr} metafunction which maps an expression to the body of its innermost-enclosing λ or the top-level program if it is not enclosed.

$$F(e, \hat{\rho}, \mathsf{mt}) = \bot \quad F(e, \hat{\rho}, \mathsf{ar}(x, e', \hat{\rho}', \hat{\kappa}')) = F(\zeta_{pr}[\![e']\!], \hat{\rho}', \hat{\kappa}') \sqcup [(e, \hat{\rho}) \mapsto \{\mathsf{ar}(x, e', \hat{\rho}')\}]$$

We now define a polymorphic refinement relation $\sqsubseteq$ over stack states and over abstract states. This relation descends componentwise: expressions, environments, and stacks each have a discrete refinement ordering (i.e. they are related only by equality); store and continuation store refinements are as follows.

$$\hat{\sigma}_a \sqsubseteq \hat{\sigma}'_a \iff \forall \hat{a} \in \widehat{Addr}_a.\hat{\sigma}_a(\hat{a}) \subseteq \hat{\sigma}'_a(\hat{a})$$
$$\tilde{\sigma}_\kappa \sqsubseteq \tilde{\sigma}'_\kappa \iff \forall \tilde{a}_\kappa \in \widetilde{KAddr}.\tilde{\sigma}_\kappa(\tilde{a}_\kappa) \subseteq \tilde{\sigma}'_\kappa(\tilde{a}_\kappa)$$

Using these definitions, stack state and abstract state refinements are as follows.

$$(e, \hat{\rho}, \hat{\sigma}_a, \hat{\kappa}) \sqsubseteq (e, \hat{\rho}, \hat{\sigma}'_a, \hat{\kappa}) \iff \hat{\sigma}_a \sqsubseteq \hat{\sigma}'_a$$
$$(e, \hat{\rho}, \hat{\sigma}_a, \tilde{\sigma}_\kappa) \sqsubseteq (e, \hat{\rho}, \hat{\sigma}'_a, \tilde{\sigma}'_\kappa) \iff \hat{\sigma}_a \sqsubseteq \hat{\sigma}'_a \text{ and } \tilde{\sigma}_\kappa \sqsubseteq \tilde{\sigma}'_\kappa$$

We now express the simulation property that constitutes soundness.

Theorem 2 (Simulation). *If $|\hat{\varsigma}| \sqsubseteq \tilde{\varsigma}$ and $\hat{\varsigma} \Rightarrow_{\hat{\Sigma}_a} \hat{\varsigma}'$, then there exists $\tilde{\varsigma}'$ such that $\tilde{\varsigma} \Rightarrow_{\tilde{\Sigma}_a} \tilde{\varsigma}'$ and $|\hat{\varsigma}'| \sqsubseteq \tilde{\varsigma}'$.*

The proof proceeds by cases on the expression, showing in each case that the abstract transition respects the relationship induced by F.

A *path* is a sequence of zero or more transitions from the initial state denoted $\hat{I}(pr) \Rightarrow^*_{\hat{\Sigma}_a} \hat{\varsigma}$. A stack $\hat{\kappa}$ is *realizable* with respect to a store $\hat{\sigma}_a$ if $(pr, \langle\rangle, \hat{\sigma}_a, \mathsf{mt}) \Rightarrow^*_{\hat{\Sigma}_a} (e, \hat{\rho}, \hat{\sigma}'_a, \hat{\kappa})$ for some expression e, environment $\hat{\rho}$, and store $\hat{\sigma}'_a$. A stack $\hat{\kappa}$ is *implied* with respect to a continuation address $(e, \hat{\rho})$ and a continuation store $\tilde{\sigma}_\kappa$, which we denote $\hat{\kappa} \in_{\tilde{\sigma}_\kappa} (e, \hat{\rho})$, as follows.

$$\mathsf{mt} \in_{\tilde{\sigma}_\kappa} (e, \hat{\rho}) \iff \mathsf{mt} \in \tilde{\sigma}_\kappa(e, \hat{\rho})$$
$$\mathsf{ar}(x, e', \hat{\rho}', \hat{\kappa}') \in_{\tilde{\sigma}_\kappa} (e, \hat{\rho}) \iff \mathsf{ar}(x, e', \hat{\rho}') \in \tilde{\sigma}_\kappa(e, \hat{\rho}) \text{ and } \hat{\kappa}' \in_{\tilde{\sigma}_\kappa} (\zeta_{pr}[\![e']\!], \hat{\rho}')$$

An empty stack is implied at an address if mt resides there. A non-empty stack is implied at an address if its top frame resides there and the remaining stack is implied by the continuation address derived from that frame. A configuration uniquely determines a continuation address, so it is sensible to consider the stacks realizable at a configuration.

Now we are able to state the precision property which, essentially, is that every reachable configuration and continuation thereat is reachable by a stack-respecting path.

Theorem 3 (Stack Precision). *Suppose $\tilde{\xi} = (\tilde{r}, \hat{\sigma}_a, \tilde{\sigma}_\kappa)$ is the least fixed point of $\Rightarrow_{\tilde{\Sigma}_a}$. For each $(e, \hat{\rho}) \in \tilde{r}$ and $\hat{\kappa}$ such that $\hat{\kappa} \in_{\tilde{\sigma}_\kappa} (\zeta_{pr}[\![e]\!], \hat{\rho})$, there exists a path $(pr, \langle\rangle, \hat{\sigma}_a, \mathsf{mt}) \Rightarrow_{\hat{\Sigma}_a} (e, \hat{\rho}, \hat{\sigma}_a, \hat{\kappa})$.*

As with Gilray *et al.* [6], the theorem is proved with two inductions, first on the path length, and second on the continuation. We omit their well-formedness property, instead relying on the supposed analysis being a *least* fixed point, which serves the same purpose to ensure that each present configuration and continuation has a reason to be. It is this property that allows the proof to easily accommodate tail calls; namely, once proper callers are ruled out as predecessors to an invocation entry, there must be a tail call which has the continuation of the entry, by definition of the continuation store tail call transition.

10 Discussion

An immediate consequence of the result that m-CFA^a is stack-precise (Theorem 3) is that m-CFA^{cps} is too, since the two analyses are isomorphic (Theorem 1).

This consequence itself is a striking result since the development of m-CFA^{cps} (1) was concurrent to and independent of the development of CFA2, the first stack-precise CFA, (2) makes no mention of stack precision, and (3) preceded P4F by more than half a decade. It also places m-CFA in a sweet spot in the CFA space, being a (1) polynomial-time, (2) stack-precise, (3) context-sensitive CFA hierarchy (4) implementable using a straightforward workset-based algorithm.

However, m-CFA exhibits additional advantages when it comes to non-local control constructs, such as exceptions, escapes, coroutines, up to full continuations. To illustrate, consider stack-precise CFAs computed by summarization

algorithms, such as CFA2. With such analyses, it is difficult to extend the analyzed language with non-local control constructs because the summarization algorithm is the sole manager of the stack. Thus, any stack-touching control feature requires the summarization algorithm to be modified in a nontrivial way. This complex algorithm lies at the heart of the analysis's soundness property, which means that such a modification requires the soundness of the analysis to be reestablished. To do this work once and for all, Vardoulakis and Shivers [22] extend the CFA2's summarization algorithm to support `call/cc`, in terms of which a host of non-local control constructs can be expressed. But, by expressing a control feature in terms of `call/cc` to obtain analysis support, one also obtains at best the precision at which `call/cc` is analyzed, and not the higher precision that weaker non-control constructs, such as exceptions and escapes, enjoy, using more-tailored modifications to the summarization algorithm [5].

In contrast, m-CFA appears to handle such contructs in its unrestricted CPS language seamlessly, with no modification to its workset algorithm, and with as much precision as current techniques. For example, when using the well-known "double-barrelled CPS" technique to encode exceptions [18], it appears that m-CFA is able to maintain perfect stack precision (also called "relative completeness" with reference to exceptions [5]) with no modification to the analysis whatsoever. We intend to formally characterize the precision m-CFA offers different continuation patterns to allow clients to engineer the CPS transformation instead of the analyzer.

The fact that m-CFA^{cps} implements P4F is due to the clever way in which Might *et al.* [13] are able to "pop" the stack of the top-m stack frames by treating the stack frame context with the discipline of a static environment—indeed, it *is* the static environment in the analysis. We can use this observation to completely isolate m-CFA's stack precision from its aggressive rebinding. That is, a variant of m-CFA which used the top-m-stack-frames context abstraction but a k-CFA-style environment would also be stack-precise (albeit exponential).

References

1. Appel, A.W.: Compiling with Continuations. Cambridge University Press (1992)
2. Felleisen, M., Friedman, D.P.: Control operators, the secd-machine, and the λ-calculus. In: Wirsing, M. (ed.) Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986. pp. 193–222. North-Holland (1987)
3. Fischer, M.J.: Lambda calculus schemata. In: Proceedings of ACM Conference on Proving Assertions About Programs, Las Cruces, New Mexico, USA, January 6-7, 1972. pp. 104–109. ACM (1972)
4. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. SIGPLAN Not. **28**(6), 237–247 (Jun 1993)
5. Germane, K., Might, M.: Relatively complete pushdown analysis of escape continuations. In: Enea, C., Piskac, R. (eds.) Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal,

January 13-15, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11388, pp. 205–225. Springer (2019)
6. Gilray, T., Lyde, S., Adams, M.D., Might, M., Van Horn, D.: Pushdown control-flow analysis for free. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 691–704. POPL '16, ACM, New York, NY, USA (Jan 2016)
7. Johnson, J.I., Van Horn, D.: Abstracting abstract control. In: Proceedings of the 10th ACM Symposium on Dynamic Languages. pp. 11–22. DLS '14, ACM, New York, NY, USA (Oct 2014)
8. Kennedy, A.: Compiling with continuations, continued. In: Hinze, R., Ramsey, N. (eds.) Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007. pp. 177–190. ACM (2007)
9. Kranz, D.A., Kelsey, R., Rees, J., Hudak, P., Philbin, J.: ORBIT: an optimizing compiler for scheme. In: Wexelblat, R.L. (ed.) Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986. pp. 219–233. ACM (1986)
10. Might, C.E.M., Horn, D.V.: Pushdown control-flow analysis of higher-order programs: Precise, polyvariant and polynomial-time. In: Scheme Workshop (2010)
11. Might, M., Shivers, O.: Environment analysis via delta CFA. In: Morrisett, J.G., Jones, S.L.P. (eds.) Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006. pp. 127–140. ACM (2006)
12. Might, M., Shivers, O.: Improving flow analyses via gammaCFA: abstract garbage collection and counting. In: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming. pp. 13–25. ICFP '06, ACM, New York, NY, USA (Sep 2006)
13. Might, M., Smaragdakis, Y., Van Horn, D.: Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 305–315. PLDI '10, ACM, New York, NY, USA (Jun 2010)
14. Palsberg, J.: Closure analysis in constraint form. ACM Trans. Program. Lang. Syst. **17**(1), 47–62 (1995)
15. Sabry, A., Felleisen, M.: Is continuation-passing useful for data flow analysis? In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation. p. 1–12. PLDI '94, Association for Computing Machinery, New York, NY, USA (1994)
16. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1991)
17. Steele Jr, G.L.: Rabbit: A compiler for Scheme. Massachusetts Institute of Technology (1978)
18. Thielecke, H.: Comparing control constructs by double-barrelled CPS. High. Order Symb. Comput. **15**(2-3), 141–160 (2002)
19. Van Horn, D., Might, M.: Abstracting abstract machines. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. pp. 51–62. ICFP '10, ACM, New York, NY, USA (Sep 2010)
20. Vardoulakis, D., Shivers, O.: CFA2: A context-free approach to control-flow analysis. In: Gordon, A.D. (ed.) Programming Languages and Systems, 19th European Symposium on Programming, ESOP 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. 6012, pp. 570–589. Springer (2010)
21. Vardoulakis, D., Shivers, O.: Ordering multiple continuations on the stack. In: Khoo, S., Siek, J.G. (eds.) Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011. pp. 13–22. ACM (2011)
22. Vardoulakis, D., Shivers, O.: Pushdown flow analysis of first-class control. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011. pp. 69–80. ACM (2011)