Relatively Complete Pushdown Analysis of Escape Continuations

Kimball Germane[1](✉) and Matthew Might[2]

[1] Brigham Young University, Provo, USA
kimball@cs.byu.edu
[2] University of Alabama, Birmingham, USA

Abstract. Escape continuations are weaker than full, first-class continuations but nevertheless can express many common control operators. Although language and compiler designs profitably leverage escape continuations, all previous approaches to analyze them statically in a higher-order setting have been ad hoc or imprecise. We present MCCFA2, a generalization of CFA2 that analyzes them with pushdown precision in their most-general form. In particular, the summarization algorithm of MCCFA2 is both sound and complete with respect to a conservative extension of CFA2's abstract semantics. We also present an *continuation age* analysis as a client of MCCFA2 that reveals critical function call optimizations.

1 Introduction

Continuations are a powerful tool in the hands of programmers, whether handled as a naked reference provided by `call/cc` or through the veneer of the exceptional `raise`, the logical `fail`, the cooperative `yield`, or the primitive `longjmp`.[1] On the other side of the language, compiler writers unify their implementations of these and other control constructs by expressing them directly via continuations [1,2,9]. While this unification has the effect of simplifying the compiler, it also amplifies the effect the compiler's power to reason about continuations has on the quality of the code it generates. Here, static analysis tools that provide maximal insight into a program's continuation use become critical.

CFA2 [19] was the first abstract interpretation of higher-order programs to model the continuation with a pushdown automaton, allowing it to precisely match calls and returns. Compared to that of finite-state models as in k-CFA [13,15], this choice of model greatly increased the precision with which continuation use could be reasoned, but at the cost of the ability to reason about any non-trivial continuation use—including any of the control constructs mentioned above. Vardoulakis and Shivers [21] extend CFA2 to soundly reason about `call/cc` but their technique sacrifices completeness w.r.t. the abstract semantics (a point we discuss further in Sect. 8.4). Vardoulakis and Shivers [21] also

[1] Of course, even `return` calls the current continuation, but we consider such uses essentially trivial.

propose two ad-hoc extensions to CFA2 to reason about exceptions. Integrating either of these proposals unduly complicates the summarization algorithm. In contrast, our approach subsumes and generalizes these proposals and yields a simpler and more coherent summarization algorithm relative to CFA2. We discuss the details of our relationship to these proposals in Sect. 11.

Although `call/cc` is a highly-expressive control construct, the power of the first-class continuations it furnishes isn't always necessary: many uses of continuations require only second-class *escape continuations*, of which `raise`, `fail`, `longjmp`, and others are thinly-masked expressions. This paper presents MCCFA2, an alternative extension to CFA2 that can reason about escape continuations both soundly and completely w.r.t. the abstract semantics and in a general, principled way.

MCCFA2 extends each of CFA2's three stages: the core language and concrete semantics, the abstract semantics, and the summarization algorithm.

1. CFA2 operates over a CPS λ-calculus statically restricted to preclude any non-trivial continuation behavior (let alone `call/cc`). We conservatively extend [4] this language to allow function calls to provide and procedures to receive and bind multiple continuations. This capability allows the language to express escape continuations generally but so regulates their lifetimes that they can be allocated on the stack [20]. To underscore this fact, our concrete semantics allocates continuations on a stack rather than a heap.
2. CFA2's abstract semantics is sound but not complete w.r.t. its concrete semantics. We extend CFA2's abstract semantics to accommodate multiple continuations. This extended abstract semantics is sound w.r.t. the extended concrete and, again, conservatively extends CFA2's. That is, MCCFA2's abstract semantics of a program in CFA2's core language are exactly as precise as CFA2's. The primary distinction between the two abstract semantics is that MCCFA2's walks the stack at each call to find the return point whereas CFA2's can determine the return point by the syntactic form of the continuation at the call site.
3. CFA2's summarization algorithm is both sound and complete w.r.t. its abstract semantics. Similarly, MCCFA2's summarization algorithm is both sound and complete w.r.t. *its* abstract semantics. To accommodate multiple continuations, MCCFA2's algorithm unifies and generalizes CFA2's by treating every continuation call as a potential escape: continuation calls that represent local returns are immediately identified as such, while those that represent non-local returns (escapes) are discovered as the algorithm walks the abstract stack.

In summary, MCCFA2 offers sound and relatively complete account of escape continuations in a general, higher-order setting. Additionally, MCCFA2 can be combined with Vardoulakis and Shivers' extension to handle first-class control which yields an analysis that forfeits precision only when continuations are used in a genuinely first-class way (Sect. 8.4).

In the next section, we discuss the MCCFA2 extension in more depth. We then establish notation (Sect. 3) and proceed to formally introduce MCPS

(Sect. 4) and its concrete (Sect. 5) and abstract (Sect. 6) semantics, connected by a sound abstraction (Sect. 7). We then present summarization (Sect. 8), by way of an algorithm (Sect. 8.2) and its correctness (Sect. 8.5). We then walk through an example MCPS program analysis (Sect. 9). Finally, we sketch how to integrate Vardoulakis and Shivers' technique to handle first-class control into MCCFA2 (Sect. 8.4) and compare MCCFA2 to other proposals to handle exceptions, as well as other related work (Sect. 11).

2 Overview

In this section, we overview MCCFA2 and discuss some significant aspects of its design.

2.1 Core Language

CFA2 considers programs to have originated in some direct-style source language before CPS conversion into its core language. Accordingly, CFA2 operates over a CPS λ-calculus partitioned into *user-world* and *continuation-world* terms [20,21]. User-world terms are those that have a direct correspondent in the source program whereas continuation-world terms are those introduced directly by the CPS transform. For instance, for a continuation reference k, the CPS term $(f\,x\,k)$ is a user-world call as it directly corresponds to the call $(f\,x)$ in the source program, whereas the CPS term $(k\,x)$ is a continuation-world call as it was synthesized from the tail-position appearance of the reference x in the source program. This static partition allows CFA2 to distinguish source-level uses of the continuation from regular function calls and thereby model such uses more precisely. CFA2's core language includes the additional restriction that continuation references may not appear free under a user-world λ-term, making it impossible to encode any control construct that interacts non-trivially with its context, let alone `call/cc`.

CFA2 is able to so precisely model the continuation behavior of the programs in its core language in part because its core language is statically limited to offer no interesting continuation behavior. MCCFA2 extends the CFA2's core language of the CPS λ-calculus to the *multiple continuation-passing style* (MCPS) λ-calculus in which function calls can provide and procedures can receive and bind multiple continuations. This ability allows the MCPS λ-calculus (or simply MCPS) to express escape continuations generally. MCPS retains the restriction that continuation references may not appear free under a user-world λ-term, which precludes it from encoding `call/cc`.

MCPS is a conservative extension [4] of single CPS that can be found in several continuation-aware compilers. For instance, MCPS limited to two continuations—"double-barreled" CPS—has been used frequently to encode exceptions and other control constructs [2,9–11,17]. MCPS is also the intermediate language of the multi-return λ-calculus (MRLC) [16] (which we revisit in Sect. 9).

The static restriction of MCPS on where continuation references may occur regulates continuation lifetimes to strictly follow a stack discipline. Thus, MCPS offers compiler writers an efficient implementation of continuations allocated on the run-time stack [7,20]. We underscore this fact by stack-allocating continuations in MCCFA2's concrete semantics (Sect. 5), deviating from CFA2's concrete semantics which heap-allocates them.

2.2 Summarization Algorithm

As it runs, CFA2's summarization algorithm records summaries of the form $(\mathit{entry}, \mathit{exit})$ which expresses that the entry state *entry* reaches the corresponding *exit* state. In the presence of multiple continuations, this form of summary doesn't adequately capture the flow of *entry* and *exit* as *exit* applies one of the multiple continuations that may be in scope at *entry*. To accommodate this fact, the summarization algorithm of MCCFA2 uses summaries of the form $(\mathit{entry}, \mathit{exit}, n)$ which include the index of the continuation (w.r.t. *entry*) called in *exit*.

In CFA2's core language, proper and tail calls in the source program are syntactically distinguished by the form of continuation: a user-world call which constructs a continuation (via a continuation-world λ-term) is a proper call whereas a user-world call which references the continuation is a tail call. CFA2's summarization algorithm exploits this knowledge by separately tracking proper callers and tail callers. When a procedure calls its continuation (i.e. *returns* in the source program), the tail callers are used to extend summaries and the proper callers are used as return points. In the presence of multiple continuations, proper and tail calls cannot in general be distinguished at the time of the call since, for the purposes of extending summaries or offering return points, the type of call is not known until the continuation is called. To accommodate this fact, the summarization algorithm of MCCFA2 (1) does not separately track proper and tail calls and (2) treats every continuation call as a potential escape. Accordingly, each continuation call instigates a phase of the algorithm which walks the abstract stack in search of return points, extending summaries as it goes.

Altogether, these changes simplify and generalize the summarization algorithm of CFA2.

3 Notation

We leverage metavariables heavily and try to be extremely careful in their use. For an arbitrary metavariable x, a bolded metavariable $\boldsymbol{x}$ represents a vector of x and a bolded, superscripted metavariable $\boldsymbol{x}^+$ represents a non-empty vector of x. The quantity $\pi_i(\boldsymbol{x})$ is the ith element of $\boldsymbol{x}$ indexed from 1. Vectors will sometimes be treated as sets and functions will sometimes be lifted over vectors. For a multi-argument function of vectors and scalars, the scalars are lifted appropriately as well. For example, $f(\boldsymbol{x}, y) = \langle f(x_1, y), \ldots, f(x_n, y)\rangle$ for $\boldsymbol{x} = \langle x_1, \ldots, x_n\rangle$. The empty vector is denoted $\langle\rangle$. We often use head–tail notation both to construct and deconstruct vectors, writing $\langle x_1, x_2, \ldots, x_n\rangle$ as $x_1 :: \langle x_2, \ldots, x_n\rangle$.

Throughout the paper, a definition's left side is a pattern which deconstructs and binds subvalues of the value of the expression on its right.

4 Partitioned CPS λ-Calculus

We view MCPS programs as being obtained by a CPS transformation of a direct-style source program. Hence, we maintain a distinction between user-world and continuation-world terms where a user-world term directly corresponds to a term in the source program and a continuation-world term is introduced by the CPS transformation. Both worlds contain λ terms, calls, and variable references. A user-world λ term *ulam* has a user parameter vector $\boldsymbol{u}$, a non-empty continuation parameter vector $\boldsymbol{k}$, and a call *call*. Given *lam* or *call*, the continuation parameter function *CP* retrieves the vector $\boldsymbol{k}$ of the innermost-enclosing *ulam* of *lam* or *call* (where a *ulam* encloses itself for this definition). A user-world call *ucall* has a user operator expression f, an argument expression vector $\boldsymbol{e}$, and a non-empty continuation expression vector $\boldsymbol{q}$. The continuation argument function *LC* retrieves the vector $\boldsymbol{q}$ of a given *ucall* or the surrounding $\boldsymbol{q}$ of a given *clam*. A user-world variable u will be bound only to user-world values. A continuation-world λ term *clam* has a user parameter vector $\boldsymbol{u}$ and a call *call*. A continuation-world call *ccall* has a continuation operator expression q and an argument expression vector $\boldsymbol{e}$. A continuation-world variable k will be bound only to continuations (Fig. 1).

All λ terms and calls are labelled uniquely in a given program to distinguish otherwise identical terms. We will sometimes identify a term with its label but the meaning should be clear from context.

Programs are closed *ulam*s with a single continuation parameter and continuation variables may not appear free within a *ulam* term.

$$pr \in Pr = \{ulam : ulam \in ULam, ulam \text{ is closed and } |CP(ulam)| = 1\}$$

$$\begin{array}{rlrrlr}
\gamma \in Lab & = \text{a set of labels} & & call \in Call & = UCall + CCall \\
u \in UVar & = \text{a set of identifiers} & & ucall \in UCall & ::= (f\, \boldsymbol{e}\, \boldsymbol{q}^{+})_\gamma \\
k \in CVar & = \text{a set of identifiers} & & ccall \in CCall & ::= (q\, \boldsymbol{e})_\gamma \\
lam \in Lam & = ULam + CLam & & e, f \in UExp & = UVar + ULam \\
ulam \in ULam & ::= (\lambda_\gamma\, (\boldsymbol{u}\, \boldsymbol{k}^{+})\; call) & & q \in CExp & = CVar + CLam \\
clam \in CLam & ::= (\lambda_\gamma\, (\boldsymbol{u})\; call) & & &
\end{array}$$

Fig. 1. Partitioned CPS λ-calculus syntax

5 Concrete Semantics

We start by defining an abstract machine to evaluate MCPS programs which will serve as the ground truth of evaluation. Like many real-world runtimes, this machine uses a stack to house both local environments and return-point information. (A stack is not necessary however; an MCPS machine that heap-allocates continuations works as well.) A value environment serves as a heap and all values are heap-allocated.

$$\begin{aligned}
\varsigma \in \mathit{State} &= \mathit{Eval} + \mathit{Apply} \\
\textsc{e} \in \mathit{Eval} &= \mathit{Call} \times \mathit{BEnv} \times \mathit{CEnv} \times \mathit{Stack} \times \mathit{VEnv} \times \mathit{Time} \\
\mathit{Apply} &= \mathit{UApply} + \mathit{CApply} \\
\textsc{ua} \in \mathit{UApply} &= \mathit{Proc} \times \boldsymbol{D} \times \boldsymbol{Cont}^{+} \times \mathit{Stack} \times \mathit{VEnv} \times \mathit{Time} \\
\textsc{ca} \in \mathit{CApply} &= \mathit{CodeP} \times \boldsymbol{D} \times \mathit{Stack} \times \mathit{VEnv} \times \mathit{Time}
\end{aligned}$$

$$\begin{aligned}
\beta_u \in \mathit{BEnv} &= \mathit{UVar} \rightharpoonup \mathit{Time} & \beta_k \in \mathit{CEnv} &= \mathit{CVar} \rightharpoonup \mathit{Cont} \\
d \in D &= \mathit{Proc} & st \in \mathit{Stack} &= \boldsymbol{Frame} \\
proc \in \mathit{Proc} &= \mathit{Clos} & fr \in \mathit{Frame} &= \mathit{BEnv} \times \mathit{CEnv} \\
\mathit{Clos} &= \mathit{ULam} \times \mathit{BEnv} & c \in \mathit{Cont} &= \mathit{CodeP} \times \mathit{FrameP} \\
ve \in \mathit{VEnv} &= \mathit{UVar} \times \mathit{Time} \rightharpoonup D & cp \in \mathit{CodeP} &= \mathit{CLam} + \{\mathit{halt}\} \\
t \in \mathit{Time} &= \text{a countably-infinite set} & fp \in \mathit{FrameP} &= \mathbb{N}
\end{aligned}$$

Fig. 2. Concrete state space

Figure 2 presents the concrete state space *State* of this machine. Each state in *State* has a stack *st*, a value environment *ve* serving as the heap, and a timestamp *t*. *State* is partitioned into two domains, *Eval* and *Apply*. An *Eval* state focuses on a call *call* in the context of a user environment β_u and continuation environment β_k. *Apply* is further partitioned into the user domain *UApply* and continuation domain *CApply*. A *UApply* state holds a procedure *proc* ready to apply to an argument vector $\boldsymbol{d}$ and non-empty continuation vector $\boldsymbol{c}$. A *CApply* state, on the other hand, holds a code pointer *cp* and result vector $\boldsymbol{d}$. We also make finer distinctions between states: an *Eval* state with a user call is a *UEval* state, denoted UE; a *Eval* state with a continuation call with a *CVar* operator is a *CEvalExit* state, denoted CEE, and a *CLam* operator is a *CEvalInner* state, denoted CEI.

Figure 3 presents the concrete machine's evaluation relation $\rightarrow$ as the union of four relations. The side conditions of each relation are divided so that user-world conditions are on the left and continuation-world on the right. (An overarching, implicit side condition is that, when a function is lifted over two different vectors, those vectors must have the same length.) The concrete machine evaluates programs by alternating between two modes: evaluating operators/arguments and applying an evaluated operator to its arguments, corresponding precisely to *Eval* and *Apply* states.

UEval

$$((f\,\boldsymbol{e}\,\boldsymbol{q}^{+})_{\gamma}, \beta_u, \beta_k, st, ve, t) \rightarrow_{UE} (proc, \boldsymbol{d}, \boldsymbol{c}, st', ve, t'), \text{ where}$$

$$\begin{aligned} proc &= \mathcal{A}_u(f, \beta_u, ve), & st' &= pop(\boldsymbol{c}, (\beta_u, \beta_k) :: st), \text{ and} \\ \boldsymbol{d} &= \mathcal{A}_u(\boldsymbol{e}, \beta_u, ve), & \boldsymbol{c} &= \mathcal{A}_k(\boldsymbol{q}, \beta_k, st) \\ t' &= tick(t), \end{aligned}$$

CEval

$$((q\,\boldsymbol{e})_{\gamma}, \beta_u, \beta_k, st, ve, t) \rightarrow_{CE} (cp, \boldsymbol{d}, st', ve, t'), \text{ where}$$

$$\begin{aligned} \boldsymbol{d} &= \mathcal{A}_u(\boldsymbol{e}, \beta_u, ve), & st' &= pop(\langle c \rangle, (\beta_u, \beta_k) :: st), \text{ and} \\ t' &= tick(t), & (cp, fp) = c &= \mathcal{A}_k(q, \beta_k, st) \end{aligned}$$

UApply

$$(proc, \boldsymbol{d}, \boldsymbol{c}, st, ve, t) \rightarrow_{UA} (call, \beta'_u, \beta_k, st, ve', t'), \text{ where}$$

$$\begin{aligned} ((\lambda_{\gamma}\,(\boldsymbol{u}\,\boldsymbol{k}^{+})\; call), \beta_u) &= proc, \\ \beta'_u &= \beta_u[\boldsymbol{u} \mapsto t'], & \beta_k &= [\boldsymbol{k} \mapsto \boldsymbol{c}] \\ ve' &= ve[(\boldsymbol{u}, t') \mapsto \boldsymbol{d}], \\ t' &= tick(t), \text{ and} \end{aligned}$$

CApply

$$((\lambda_{\gamma}\,(\boldsymbol{u})\; call), \boldsymbol{d}, (\beta_u, \beta_k) :: st, ve, t) \rightarrow_{CA} (call, \beta'_u, \beta_k, st, ve', t'), \text{ where}$$

$$\begin{aligned} \beta'_u &= \beta_u[\boldsymbol{u} \mapsto t'], \\ ve' &= ve[(\boldsymbol{u}, t') \mapsto \boldsymbol{d}], \text{ and} \\ t' &= tick(t) \end{aligned}$$

Argument Evaluation

$$\begin{aligned} \mathcal{A}_u(u, \beta_u, ve) &= ve(u, \beta_u(u)) & \mathcal{A}_k(k, \beta_k, st) &= \beta_k(k) \\ \mathcal{A}_u(ulam, \beta_u, ve) &= (ulam, \beta_u) & \mathcal{A}_k(clam, \beta_k, st) &= (clam, |st| + 1) \end{aligned}$$

pop **Metafunction**

$$pop(\boldsymbol{c}, st) = st|_{\max(height(\boldsymbol{c}))} \text{ where } height(cp, fp) = fp$$

Program Injection

$$\mathcal{I}(pr, \boldsymbol{d}) = ((pr, \bot), \boldsymbol{d}, \langle (halt, 0) \rangle, \langle\rangle, \bot, t_0)$$

Fig. 3. The concrete semantics

For *Eval* states, the metafunction $\mathcal{A}_u$ evaluates atomic user-world expressions, dereferencing variables and constructing closures. Likewise, $\mathcal{A}_k$ evaluates continuation expressions, dereferencing variables and constructing code–frame pointer pairs. As part of each call—user or continuation—the evaluated continuations are used to determine the youngest live frame on the stack. If a call doesn't reference some continuations in scope, the frames unique to them become dead. The *pop* metafunction discards all such frames that reside at the top of the stack as part of the transition (where the notation $st|_{fp}$ indicates the oldest *fp* frames of *st*). Hence, arbitrarily many frames may be popped when a call is made. This stack management policy follows Might and Shivers' generalization [14] of the stack management policy of the Orbit compiler [1].

Apply states precipitate the procedure entry or re-entry, depending on whether a user procedure or continuation is applied. Application of a user procedure *proc* extends its environment β_u with bindings for the arguments and the heap *ve* with their values, as well as constructing a continuation environment β_k. New user bindings use the timestamp t' to ensure freshness; in this work, this is the sole use of timestamps. Continuation application entails popping the stack to *c*'s frame pointer *fp*, jumping to *c*'s code pointer *cp*, and extending the local environment β_u with bindings and the heap *ve* with the result values.

A program *pr* and its arguments $\boldsymbol{d}$ are injected into a *UApply* state with a single *halt* continuation pointing to the base of the stack, empty stack, empty value environment, and epoch time.

6 Abstract Semantics

The next stage of MCCFA2 is the definition of an abstract semantics. The abstract state space $\widehat{State}$, seen in Fig. 4, is partitioned identically to *State*. However, abstract states themselves and their components differ nontrivially from their concrete counterparts. Following CFA2, abstract states lack timestamps,

$$\begin{aligned}
\hat{\varsigma} \in \widehat{State} &= \widehat{Eval} + \widehat{Apply} \\
\hat{\text{E}} \in \widehat{Eval} &= Call \times \widehat{Stack} \times Heap \\
\widehat{Apply} &= \widehat{UApply} + \widehat{CApply} \\
\hat{\text{UA}} \in \widehat{UApply} &= \widehat{Proc} \times \widehat{\boldsymbol{D}} \times \widehat{\boldsymbol{CExp}}^{+} \times \widehat{Stack} \times Heap \\
\hat{\text{CA}} \in \widehat{CApply} &= CodeP \times \widehat{\boldsymbol{D}} \times \widehat{Stack} \times Heap
\end{aligned}$$

$$\begin{aligned}
\widehat{proc} \in \widehat{Proc} &= \widehat{Clos} \\
\hat{d} \in \widehat{D} &= \mathcal{P}(\widehat{Proc}) \\
\widehat{Clos} &= ULam \\
h \in Heap &= UVar \rightharpoonup \widehat{D}
\end{aligned}
\qquad
\begin{aligned}
\widehat{st} \in \widehat{Stack} &= \widehat{\boldsymbol{Frame}} \\
\widehat{Frame} &= SyntaxMap \\
sm \in SyntaxMap &= CVar \rightharpoonup \widehat{CExp} \\
\hat{q} \in \widehat{CExp} &= CExp + \{halt\}
\end{aligned}$$

Fig. 4. Abstract state space

$\widehat{UEval}$

$$((f\,\boldsymbol{e}\,\boldsymbol{q}^{+})_{\gamma}, \widehat{st}, h) \leadsto_{UE} (ulam, \hat{\boldsymbol{d}}, \hat{\boldsymbol{q}}, \widehat{st}', h), \text{ where}$$

$$ulam \in \hat{\mathcal{A}}(f,h), \qquad (\hat{\boldsymbol{q}}, \widehat{st}') = \widehat{pop}(\boldsymbol{q}, \widehat{st})$$
$$\hat{\boldsymbol{d}} = \hat{\mathcal{A}}(\boldsymbol{e},h), \text{ and}$$

$\widehat{CEval}$

$$((q\,\boldsymbol{e})_{\gamma}, \widehat{st}, h) \leadsto_{CE} (cp, \hat{\boldsymbol{d}}, \widehat{st}', h), \text{ where}$$

$$\hat{\boldsymbol{d}} = \hat{\mathcal{A}}(\boldsymbol{e},h) \text{ and} \qquad (\langle cp\rangle, \widehat{st}') = \widehat{pop}(\langle q\rangle, \widehat{st})$$

$\widehat{UApply}$

$$((\lambda_{\gamma}\,(\boldsymbol{u}\,\boldsymbol{k}^{+})\;call), \hat{\boldsymbol{d}}, \hat{\boldsymbol{q}}, \widehat{st}, h) \leadsto_{UA} (call, \widehat{st}', h'), \text{ where}$$

$$h' = h \sqcup [\boldsymbol{u} \mapsto \hat{\boldsymbol{d}}] \text{ and} \qquad \widehat{st}' = [\boldsymbol{k} \mapsto \hat{\boldsymbol{q}}] :: \widehat{st}$$

$\widehat{CApply}$

$$((\lambda_{\gamma}\,(\boldsymbol{u})\;call), \hat{\boldsymbol{d}}, \widehat{st}, h) \leadsto_{CA} (call, \widehat{st}, h'), \text{ where}$$

$$h' = h \sqcup [\boldsymbol{u} \mapsto \hat{\boldsymbol{d}}]$$

Argument Evaluation

$$\hat{\mathcal{A}}(ulam, h) = \{ulam\} \qquad \hat{\mathcal{A}}(u,h) = h(u)$$

$\widehat{pop}$ **Metafunction**

$$\widehat{pop}(\boldsymbol{k}, sm :: \widehat{st}') = \widehat{pop}(sm(\boldsymbol{k}), \widehat{st}')$$
$$\widehat{pop}(\hat{\boldsymbol{q}}, \widehat{st}) = (\hat{\boldsymbol{q}}, \widehat{st}) \text{ if } \pi_i(\hat{\boldsymbol{q}}) \in \mathit{CodeP} \text{ for some } i$$

Program Injection

$$\hat{\mathcal{I}}(pr, \hat{\boldsymbol{d}}) = (pr, \hat{\boldsymbol{d}}, \langle halt\rangle, \langle\rangle, \bot)$$

Fig. 5. The abstract semantics

an abstract denotable $\hat{d}$ is a superposition of procedures $\widehat{proc}$; and closures no longer include an environment.[2]

MCCFA2 summarization crucially relies on a non-standard abstraction for continuation environments: the *syntax map*. A syntax map is a finite mapping from continuation variables to continuation expression syntax, i.e., the syntax of the continuation arguments in a call expression. Each stack frame houses a syntax map and a stack of these frames comprises the program's control linkage information. The particular maintenance of this stack, which we describe shortly, allows us to omit frame pointers from continuations—they consist of merely a code pointer.

[2] Deviating from CFA2, we omit environments from stack frames as well. This is only to simplify the presentation; they can be reintroduced without difficulty.

We define the abstract semantics as a union of the four relations defined in Fig. 5. Once again, user-world conditions are sequestered to the left column and continuation-world to the right. Because multiple procedure values may be in superposition as a call's operator, the target procedure is chosen nondeterministically. Just as in the concrete semantics, the stack is popped at both user and continuation calls according to the generalized Orbit policy. However, the mechanism by which this policy is upheld is significantly different.

When a user call is made, the $\widehat{pop}$ metafunction uses the call's continuation expressions $\boldsymbol{q}$ and the stack $\widehat{st}$ to determine the dead frames (if any) to pop from the stack. If there is some *clam* within $\boldsymbol{q}$, then the caller frame is live (since a nested call may return to it) and no dead frames can be popped. On the other hand, if $\boldsymbol{q} = \boldsymbol{k}$ for some $\boldsymbol{k}$, the call is a tail call and at least one dead frame (the caller's) can be popped. Thus, each step of $\widehat{pop}$ determines whether the top frame of the provided stack is dead and, if so, pops it. It may be that such a step results in some syntax vector $\boldsymbol{k}'$ which indicates that the newly-revealed top frame is dead also, a situation that occurs when a call doesn't reference the only continuations on which multiple frames atop the stack depend. For this reason, $\widehat{pop}$ is recursive and can pop arbitrarily-many frames in a given transition.

The $\widehat{pop}$ metafunction is used to implement continuation calls as well. When used in this way, the continuation operator q is provided to $\widehat{pop}$ along with the stack. If $q \in \mathit{CLam}$, the call represents a `let`-continuation, a local binding construct in the source program. In this case, $\widehat{pop}$ correctly determines that the top frame is live and preserves the stack. If $q \in \mathit{CVar}$, the call represents a return to some continuation in scope. In this case, the recursive definition of $\widehat{pop}$ effects the popping of the stack and discovery of the return point.

The $\hat{\mathcal{I}}$ metafunction injects a program and abstract argument vector into an initial machine state.

7 Abstraction

With machines for both the concrete and abstract semantics defined, we need to ensure that the abstract semantics *simulates* the concrete semantics. To obtain this assurance, we first need to establish a correspondence between their state spaces and introduce a notion of precision into the abstract state space. Figure 6 presents this correspondence via the concrete–abstract abstraction map $|\cdot|_{ca}$ and the abstraction refinement relation $\sqsubseteq$.

The bulk of $|\cdot|_{ca}$ is contained in the mutually-inductive metafunctions *reconstruct* and *reconstruct** which reconstruct an abstract stack and continuations from a concrete stack and continuations. If the stack *st* given to *reconstruct** is empty, the given argument vector $\boldsymbol{c}$ must be **halt** and the result is its abstraction **halt** paired with the empty stack. Otherwise, the code pointer *clam* of the *height*-maximum continuation c is determined and the continuation argument syntax vector $\boldsymbol{q}$ in which it's found is paired with the *reconstruct*ion of the continuation parameter vector $\boldsymbol{k}$ of its enclosing λ-term and the rest of the stack. The *reconstruct* metafunction uses *reconstruct** to reconstruct all but the top

Abstraction

$$|((ulam,\beta_u),\boldsymbol{d},\boldsymbol{c},st,ve,t)|_{ca} = (ulam,|\boldsymbol{d}|_{ca},\hat{\boldsymbol{q}},\widehat{st},|ve|_{ca}) \text{ where } (\hat{\boldsymbol{q}},\widehat{st}) = reconstruct^*(\boldsymbol{c},st)$$

$$|(cp,\boldsymbol{d},st,ve,t)|_{ca} = (cp,|\boldsymbol{d}|_{ca},\widehat{st},|ve|_{ca}) \text{ where } (\langle cp\rangle,\widehat{st}) = reconstruct^*(\langle(cp,|st|)\rangle,st)$$

$$|(call,\beta_u,\beta_k,st,ve,t)|_{ca} = (call,\widehat{st},|ve|_{ca}) \text{ where } \widehat{st} = reconstruct(CP(call),\beta_k,st)$$

$$|(ulam,\beta_u)|_{ca} := \{ulam\}$$
$$|(cp,fp)|_{ca} := cp$$
$$|ve|_{ca} := \{(u,\bigsqcup_t |ve(u,t)|_{ca})\}$$

Stack Reconstruction

$$reconstruct^*(\boldsymbol{c},st) = \begin{cases}(|\boldsymbol{c}|_{ca},\langle\rangle) \text{ if } st=\langle\rangle\\ (\boldsymbol{q}, reconstruct(\boldsymbol{k},\beta_k,st')) \text{ if } st = (\beta_u,\beta_k) :: st'\end{cases}$$

where $\boldsymbol{k} = CP(clam)$ and $\boldsymbol{q} = LC(clam)$ for $(clam, fp) = \text{argmax}(height(\boldsymbol{c}))$

$$reconstruct(\boldsymbol{k},\beta_k,st) = sm :: \widehat{st} \text{ where } sm = [\boldsymbol{k}\mapsto\hat{\boldsymbol{q}}] \text{ and } (\hat{\boldsymbol{q}},\widehat{st}) = reconstruct^*(\beta_k(\boldsymbol{k}),st)$$

Refinement

$$(ulam,\hat{\boldsymbol{d}}_1,\hat{\boldsymbol{q}},\widehat{st},h_1) \sqsubseteq (ulam,\hat{\boldsymbol{d}}_2,\hat{\boldsymbol{q}},\widehat{st},h_2) \text{ iff } \hat{\boldsymbol{d}}_1 \sqsubseteq \hat{\boldsymbol{d}}_2 \text{ and } h_1 \sqsubseteq h_2$$
$$(cp,\hat{\boldsymbol{d}}_1,\widehat{st},h_1) \sqsubseteq (cp,\hat{\boldsymbol{d}}_2,\widehat{st},h_2) \text{ iff } \hat{\boldsymbol{d}}_1 \sqsubseteq \hat{\boldsymbol{d}}_2 \text{ and } h_1 \sqsubseteq h_2$$
$$(call,\widehat{st},h_1) \sqsubseteq (call,\widehat{st},h_2) \text{ iff } h_1 \sqsubseteq h_2$$

$$\langle\hat{d}_{1,1},\ldots,\hat{d}_{1,n}\rangle \sqsubseteq \langle\hat{d}_{2,1},\ldots,\hat{d}_{2,n}\rangle \text{ iff } \hat{d}_{1,i} \sqsubseteq \hat{d}_{2,i} \text{ for } i=1,\ldots,n$$

$$\hat{d}_1 \sqsubseteq \hat{d}_2 \text{ iff } \hat{d}_1 \subseteq \hat{d}_2$$

Fig. 6. Abstraction map and refinement relation

frame of the stack. It uses the given continuation parameter vector $\boldsymbol{k}$ and the resultant continuation argument syntax vector $\hat{\boldsymbol{q}}$ to build the top frame.

The abstraction refinement relation $\sqsubseteq$ is standard.

Theorem 1 (Simulation).
If $\varsigma \to \varsigma'$ and $|\varsigma|_{ca} \sqsubseteq \hat{\varsigma}$, then there exists $\hat{\varsigma}'$ such that $\hat{\varsigma} \leadsto \hat{\varsigma}'$ and $|\varsigma'|_{ca} \sqsubseteq \hat{\varsigma}'$.

The soundness of the *Eval–Apply* transitions are non-trivial as they must establish that equivalent frames are popped from the stack in the transitions. To establish it, we use the following lemma.

Lemma 1. *Suppose $|\text{UE}|_{ca} \sqsubseteq \hat{\text{UE}}$ where $\text{UE} = ((f\ \boldsymbol{e}\ \boldsymbol{q}^+)_\gamma,\beta_u,\beta_k,st,ve,t)$. If $\boldsymbol{k} = CP(\gamma)$, $\mathcal{A}_k(\boldsymbol{q},\beta_u,st) = \boldsymbol{c}$, $reconstruct(\boldsymbol{k},\beta_k,st) = \widehat{st}$, and $pop(\boldsymbol{c},(\beta_u,\beta_k) :: st) = st'$, then $reconstruct^*(\boldsymbol{c},st') = \widehat{pop}(\boldsymbol{q},\widehat{st})$.*

This lemma establishes that *pop* and $\widehat{pop}$ commute with *reconstruct* and *reconstruct**. That is, given a concrete stack *st*, one can *reconstruct* an abstract stack $\widehat{st}$ and $\widehat{pop}$ it to $\widehat{st}'$ or *pop* it to st' and *reconstruct** to obtain $\widehat{st}'$. This lemma is established by proving that *reconstruct*/*reconstruct** yield stacks that preserve the behavior of *pop* in $\widehat{pop}$. A proof is given in a technical report [5].

$$\tilde{\varsigma} \in \widetilde{State} = \widetilde{Eval} + \widetilde{Apply}$$

$$\begin{aligned} \widetilde{Eval} &= \widetilde{UEval} + \widetilde{CEval} & \widetilde{Apply} &= \widetilde{UApply} + \widetilde{CApply} \\ \tilde{\text{UE}} \in \widetilde{UEval} &= UCall \times Heap & \tilde{\text{UA}} \in \widetilde{UApply} &= \widehat{UProc} \times \widehat{D}^* \times Heap \\ \tilde{\text{CE}} \in \widetilde{CEval} &= CCall \times Heap & \tilde{\text{CA}} \in \widetilde{CApply} &= \widehat{CProc} \times \widehat{D}^* \times Heap \end{aligned}$$

Fig. 7. Local state space

8 Summarization

Because abstract stacks are unbounded, the abstract state space is infinite. Hence, we can't perform abstract interpretation simply by enumerating the states reachable from the program entry state. Instead, we'll perform it using a summarization algorithm similar to that of CFA2.

Summarization algorithms are so-called because they discover and summarize reachability facts between system states. CFA2's summarization algorithm summarizes the fact that *exit* is reachable from *entry* in a stack-respecting way with a pair $(entry, exit)$. This form of summary is inadequate for MCPS. Instead, we use a summary $(entry, exit, n)$ to record the fact that both *exit* is reachable from *entry* in a stack-respecting way and *exit* is returning to the nth continuation of *entry*.

In Sect. 8.5, we show that the summarization algorithm inherently respects the stack. Consequently, the stack component of abstract states is unnecessary and the summarization algorithm operates over the *local semantics*, the stack-free residue of the abstract semantics.

$$((f\, \boldsymbol{e}\, \boldsymbol{q}^+)_\ell, h) \approx\!> _{UE} (ulam, \hat{\boldsymbol{d}}, h)$$

$$ulam \in \hat{\mathcal{A}}(f, h)$$
$$\hat{\boldsymbol{d}} = \hat{\mathcal{A}}(\boldsymbol{e}, h)$$

$$((clam\, \boldsymbol{e}), h) \approx\!> _{CE} (clam, \hat{\boldsymbol{d}}, h)$$

$$\hat{\boldsymbol{d}} = \hat{\mathcal{A}}(\boldsymbol{e}, h)$$

$$((\lambda_\gamma\, (\boldsymbol{u}\, \boldsymbol{k}^+)\ call), \hat{\boldsymbol{d}}, h) \approx\!> _{UA} (call, h')$$
$$h' = h \sqcup [\boldsymbol{u} \mapsto \hat{\boldsymbol{d}}]$$

$$((\lambda_\gamma\, (\boldsymbol{u})\ call), \hat{\boldsymbol{d}}, h) \approx\!> _{CA} (call, h')$$
$$h' = h \sqcup [\boldsymbol{u} \mapsto \hat{\boldsymbol{d}}]$$

Fig. 8. The local semantics

8.1 Local Semantics

The local semantics describes segments of evaluation that don't require the return-point information of the stack. Figure 7 contains the local state space, which is simply the abstract state space with stacks excised. Accordingly, the local abstraction map $|\cdot|_{al}$ merely performs the excision:

```
$Summary, Call, Final = \emptyset$
$Seen, Work = \{(\tilde{\mathcal{I}}(pr, \hat{\boldsymbol{d}}), \tilde{\mathcal{I}}(pr, \hat{\boldsymbol{d}}))\}$

while $Work \neq \emptyset$:
  remove $(\tilde{\text{UA}}, \tilde{\varsigma})$ from $Work$
  switch $\tilde{\varsigma}$:
    case $\tilde{\text{UA}}, \tilde{\text{CA}}, \tilde{\text{CEI}}$:
      for $\tilde{\varsigma}' \in succ(\tilde{\varsigma})$:
        Propagate$(\tilde{\varsigma}, \tilde{\varsigma}')$
    case $\tilde{\text{UE}}$:
      for $\tilde{\varsigma}' \in succ(\tilde{\varsigma})$:
        Propagate$(\tilde{\varsigma}', \tilde{\varsigma}')$
        insert $(\tilde{\text{UA}}, \tilde{\varsigma}, \tilde{\varsigma}')$ into $Call$
        for $(\tilde{\varsigma}', \tilde{\text{CEE}}, n) \in Summary$:
          Link$(\tilde{\text{UA}}, \tilde{\varsigma}, \tilde{\varsigma}', \tilde{\text{CEE}}, n)$
    case $\tilde{\text{CEE}}$:
      Return$(\tilde{\text{UA}}, \tilde{\varsigma}, CP(\tilde{\text{UA}}, CV(\tilde{\text{CEE}})))$

Propagate$(\tilde{\text{UA}}, \tilde{\varsigma})$:=
  if $(\tilde{\text{UA}}, \tilde{\varsigma}) \notin Seen$:
    insert $(\tilde{\text{UA}}, \tilde{\varsigma})$ into $Seen$
    insert $(\tilde{\text{UA}}, \tilde{\varsigma})$ into $Work$

Return$(\tilde{\text{UA}}, \tilde{\text{CEE}}, n)$:=
  if $(\tilde{\text{UA}}, \tilde{\text{CEE}}, n) \notin Summary$:
    insert $(\tilde{\text{UA}}, \tilde{\text{CEE}}, n)$ into $Summary$
    if $\tilde{\text{UA}} = \tilde{\mathcal{I}}(pr, \hat{\boldsymbol{d}})$:
      Final$(\tilde{\text{CEE}})$
    for $(\tilde{\text{UA}}^*, \tilde{\text{UE}}, \tilde{\text{UA}}) \in Call$:
      Link$(\tilde{\text{UA}}^*, \tilde{\text{UE}}, \tilde{\text{UA}}, \tilde{\text{CEE}}, n)$

Link$(\tilde{\text{UA}}, \tilde{\text{UE}}, \tilde{\text{UA}}^*, \tilde{\text{CEE}}, n)$:=
  switch $CA(\tilde{\text{UE}}, n)$:
    case $k$:
      Return$(\tilde{\text{UA}}, \tilde{\text{CEE}}, CP(\tilde{\text{UA}}, k))$
    case $clam$:
      Update$(\tilde{\text{UA}}, clam, \tilde{\text{CEE}},,$
)
Update$(\tilde{\text{UA}}, clam, \tilde{\text{CEE}},, :=)$
  $\tilde{\text{CEE}}$ of form $((k\, \boldsymbol{e}), h)$
  $\hat{\boldsymbol{d}} = \mathcal{A}_u(\boldsymbol{e}, h)$
  Propagate$(\tilde{\text{UA}}, (clam, \hat{\boldsymbol{d}}, h))$

Final$(\tilde{\varsigma})$:=
  $\tilde{\varsigma}$ of form $((k\, \boldsymbol{e}), h)$
  insert $(halt, \mathcal{A}_u(\boldsymbol{e}, h), h)$ into $Final$
```

Fig. 9. The summarization algorithm

$$|(ulam, \hat{\boldsymbol{d}}, \hat{\boldsymbol{q}}, \widehat{st}, h)|_{al} = (ulam, \hat{\boldsymbol{d}}, h)$$
$$|(clam, \hat{\boldsymbol{d}}, \widehat{st}, h)|_{al} = (clam, \hat{\boldsymbol{d}}, h)$$
$$|(call, \widehat{st}, h)|_{al} = (call, h)$$

We define the local semantics as the union of four relations over local states, seen in Fig. 8. This semantics is similar to the abstract semantics except that it is not defined over continuation calls that exit the procedure, which requires the return-point information of the stack. The summarization algorithm is tasked with linking exits to their return points.

The *local successors* $succ(\tilde{\varsigma})$ of a state $\tilde{\varsigma}$ is defined $succ(\tilde{\varsigma}) = \{\tilde{\varsigma}' : \tilde{\varsigma} \approx\!> \tilde{\varsigma}'\}$.

8.2 Summarization Algorithm

Figure 9 presents the summarization algorithm. The product of running the algorithm is three relations: the ternary relations *Summary* and *Call*, and the unary relation *Final*. A summary $(\tilde{\text{UA}}, \tilde{\text{CEE}}, n) \in Summary$ records the fact that $\tilde{\text{CEE}}$ exits the procedure entered by $\tilde{\text{UA}}$ through its nth continuation (by position). A call edge $(\tilde{\text{UA}}_0, \tilde{\text{UE}}, \tilde{\text{UA}}) \in Call$ records the fact that, in the invocation $\tilde{\text{UA}}_0$ heads, the call $\tilde{\text{UE}}$ yields the entry $\tilde{\text{UA}}$. A state $\tilde{\text{UA}} \in Final$ is simply a terminal state of evaluation.

We define the summarization algorithm imperatively and based on a workset, after the style of CFA2's. The workset consists of pairs of states of form $(\tilde{\text{UA}}, \tilde{\varsigma})$ where $\tilde{\text{UA}}$ is entry state of the procedure invocation containing $\tilde{\varsigma}$. The workset is initialized with $\tilde{\mathcal{I}}(pr, \hat{\boldsymbol{d}})$ paired with itself (where $\tilde{\mathcal{I}}(pr, \hat{\boldsymbol{d}}) = |\hat{\mathcal{I}}(pr, \hat{\boldsymbol{d}})|_{al}$).

When $\tilde{\varsigma}$ has local successors (as determined by $\widetilde{succ}$), these are `Propagate`d to the workset. When $\tilde{\varsigma} \in \widetilde{UEval}$, its successors are $\widetilde{UApply}$ states which are their own corresponding entries. Thus, each successor $\tilde{\varsigma}'$ is `Propagate`d with itself and the call is recorded in *Call*. Each summary that begins with $\tilde{\varsigma}'$ is `Link`ed with the caller. `Link` searches for the return point of a call by looking at the continuation expression at the position the summary exited. If that continuation expression is some k, `Return` searches deeper in the stack with that continuation position mapped through the formal parameters. If that continuation expression is some *clam*, `Update` synthesizes the return point and `Propagate`s it.

Finally, when $\tilde{\varsigma} \in \widetilde{CEvalExit}$, its continuation position with respect to $\tilde{\text{UA}}$ is determined by *CP* and passed to `Return`. If the entry–exit–position triple is already recorded in *Summary*, the path is sure to be explored and the search is cut off. Otherwise, the triple is recorded in *Summary*. If $\tilde{\text{UA}}$ is the program entry state, then $\tilde{\varsigma}$ is a program exit state and `Final` uses it to synthesize a state to record in *Final*. The last step of `Return` `Link`s every caller of $\tilde{\text{UA}}$ with the triple.

8.3 Comparison with CFA2

The MCCFA2 summarization algorithm simplifies and generalizes that of CFA2.

To simplify, MCCFA2's algorithm builds a single *Call* relation where CFA2 builds the *Callers* relation for proper callers and *TCallers* relation for tail callers. Our consolidation of these relations was due to expediency: with multiple continuations, one can't in general determine whether a call's particular continuation will be invoked at the point of the call and hence the type of call cannot be known a priori. However, the result is a more uniform treatment of calls which is both simpler and more general.

MCCFA2's algorithm also operates over a more general language than CFA2's—the MCPS λ-calculus. The presence of multiple continuations means the return point of a call is no longer guaranteed to be at the top of the stack. To reflect this, MCCFA2's algorithm essentially has two phases: the first drives the workset loop and explores the state space; the second is activated when a procedure exits and the stack is walked to find the return point.

8.4 Integrating First-Class Control

Vardoulakis and Shivers [21] extend CFA2 to handle `call/cc` by allowing free continuation references in operator position. The similarly-extended summarization algorithm keeps track of two additional unary relations: *EntriesEsc* contains entry states of procedures that bind escaping continuations (that is, continuations with free references) and *Escapes* contains exit states in which escaped continuations are applied. When the algorithm encounters an entry state over a procedure *ulam* that binds an escaping continuation k, that state is added to *EntriesEsc* and linked (by summary) to any *Escapes* states that apply k. On the other end, when it encounters an exit state that applies an escaped continuation bound to k, that state is added to *Escapes* and linked to any *EntriesEsc* states that bind k.

Because the non-local linking it performs ignores path realizability, the extended summarization algorithm is incomplete with respect to the CFA2's abstract semantics. Thus, summarization introduces spurious paths for even morally second-class uses of `call/cc`, such as exceptions. The present work has demonstrated that such uses can be treated completely with respect to the abstract semantics. Furthermore, we can integrate this extension into MCCFA2 and pay-as-we-go for analysis of bonafide first-class control but enjoy complete analysis otherwise.

To integrate this technique, we also keep track of *EntriesEsc* and *Escapes*. We add an entry state to *EntriesEsc* when any of the continuations it binds escape and link it to *Escapes* states that invoke any escaped continuation under those bound names. We add an exit state to *Escapes* when it applies an escaped continuation and link it to *EntriesEsc* states that bind the continuation's name. We include the binding continuation of the escaped continuation in the synthesized link to let `Return` propagate the control transfer.

In Sect. 11, we consider specific extensions to CFA2 and how this work subsumes or enhances them.

8.5 Summarization Correctness

Our summarization algorithm is sound and complete with respect to the abstract semantics. Before we formally define those properties, we need to introduce some auxiliary definitions.

A *path* p is a sequence of abstract states $\hat{\varsigma}_0, \hat{\varsigma}_1, \ldots, \hat{\varsigma}_n$ where $\hat{\varsigma}_0 \leadsto \hat{\varsigma}_1 \leadsto \ldots \leadsto \hat{\varsigma}_n$. We denote by $p_0 \leadsto p_1$ the concatenation of paths p_0 and p_1. The smallest reflexive relation over $\widehat{State}$ is denoted by $\leadsto^0$, the transitive closure of $\leadsto$ by $\leadsto^+$, and the reflexive, transitive closure of $\leadsto$ by $\leadsto^*$.

To extract the continuation variable from $\hat{\text{CEE}} = ((k\,\boldsymbol{e})_\gamma, \widehat{st}, h)$, let $CV(\hat{\text{CEE}}) = k$. To determine the continuation position of a continuation from the operator of $\hat{\text{UA}} = ((\lambda_\gamma\,(\boldsymbol{u}\,\boldsymbol{k}^+)\,call), \hat{\boldsymbol{d}}, \hat{\boldsymbol{q}}, \widehat{st}, h)$, let $CP(\hat{\text{UA}}, k) = i$ where $\pi_i(\boldsymbol{k}) = k$. Finally, to extract the ith continuation argument (by position) from $\hat{\text{UE}} = ((f\,\boldsymbol{e}\,\boldsymbol{q}^+)_\gamma, \widehat{st}, h)$, let $CA(\hat{\text{UE}}, i) = \pi_i(\boldsymbol{q})$.

The *corresponding entry* of an abstract state is the entry state of the invocation of which it's a part.

Definition 1 (Corresponding Entry). *Let $CE_p(\hat{\varsigma})$ denote the* corresponding entry *of a state $\hat{\varsigma}$ in path p. For path $p \equiv \hat{\text{UA}} \leadsto^* \hat{\varsigma}$, $CE_p(\hat{\varsigma}) = \hat{\text{UA}}$ if:*

1. $p \equiv \hat{\text{UA}} \leadsto^0 \hat{\varsigma}$;

2. $p \equiv \hat{\text{UA}} \leadsto^* \hat{\varsigma}' \leadsto \hat{\varsigma}$, $\hat{\text{UA}} = CE_p(\hat{\varsigma}')$, $\hat{\varsigma}' \notin \widehat{UEval}$, and $\hat{\varsigma}' \notin \widehat{CEvalExit}$; or

3. *$p \equiv \hat{\text{UA}} \rightsquigarrow^+ \hat{\text{UE}} \rightsquigarrow \hat{\text{UA}}' \rightsquigarrow^+ \hat{\text{CEE}} \rightsquigarrow \hat{\varsigma}$, $\hat{\text{UA}} = CE_p(\hat{\text{UE}})$, $CA(\hat{\text{UE}}, n) \in CLam$, and $\hat{\text{UA}}' \equiv_p \hat{\text{CEE}}$ by n.*

For a path $p \equiv \hat{\text{UA}} \rightsquigarrow^+ \hat{\text{CEE}}$, we say $\hat{\text{UA}} \equiv_p \hat{\text{CEE}}$ by n if:

1. *$\hat{\text{UA}} = CE_p(\hat{\text{CEE}})$ and $CP(\hat{\text{UA}}, CV(\hat{\text{CEE}})) = n$; or*
2. *$p \equiv \hat{\text{UA}} \rightsquigarrow^+ \hat{\text{UE}} \rightsquigarrow \hat{\text{UA}}' \rightsquigarrow^+ \hat{\text{CEE}}$, $\hat{\text{UA}} = CE_p(\hat{\text{UE}})$, $\hat{\text{UA}}' \equiv_p \hat{\text{CEE}}$ by n', $CA(\hat{\text{UE}}, n') = k$, and $CP(\hat{\text{UA}}, k) = n$.*

The first case of CE_p says that, in path p, a procedure entry state $\hat{\text{UA}}$ is its own corresponding entry. The second says that the corresponding entry is preserved across an intraprocedural transition. The third says that a return state $\hat{\text{CA}}$ has the corresponding entry of a user call state $\hat{\text{UE}}$ if its nth continuation argument is some *clam* and that $\hat{\text{UA}}' \equiv_p \hat{\text{CEE}}$ by n holds for the intervening path $\hat{\text{UA}}' \rightsquigarrow^+ \hat{\text{CEE}}$. Two states with the same corresponding entry are part of the same abstract procedure invocation.

The ternary "same-level" relation $\cdot \equiv_p \cdot$ by $\cdot$ captures the fact that an exit state $\hat{\text{CEE}}$ returns through a sequence of tail calls through $\hat{\text{UA}}$'s nth continuation. The base case relates an entry state $\hat{\text{UA}}$ and an exit state $\hat{\text{CEE}}$ in the same invocation that returns through the nth continuation of $\hat{\text{UA}}$. The inductive case assumes that $\hat{\text{CEE}}$ returns through the n'th continuation of $\hat{\text{UA}}'$ and, if the n'th continuation of its caller $\hat{\text{UE}}$ is a reference, extends it by $\hat{\text{UE}}$'s corresponding entry $\hat{\text{UA}}$ and the position n of the referenced continuation with respect to $\hat{\text{UA}}$.

Summarization Soundness. Soundness is the property that any abstract path p initiated by $\hat{\mathcal{I}}(pr, \hat{\boldsymbol{d}})$ is contained in the results of summarization. Formally, we have the following:

Theorem 2 (Soundness).
After summarization,

1. *if $p \equiv \hat{\mathcal{I}}(pr, \hat{\boldsymbol{d}}) \rightsquigarrow^* \hat{\text{UA}} \rightsquigarrow^+ \hat{\text{CEE}}$ such that $\hat{\text{UA}} \equiv_p \hat{\text{CEE}}$ by n, then $(|\hat{\text{UA}}|_{al}, |\hat{\text{CEE}}|_{al}, n) \in Summary$; and*
2. *if $p \equiv \hat{\mathcal{I}}(pr, \hat{\boldsymbol{d}}) \rightsquigarrow^* \hat{\text{UA}} \rightsquigarrow^+ \hat{\text{UE}} \rightsquigarrow \hat{\text{UA}}'$ such that $\hat{\text{UA}} = CE_p(\hat{\text{UE}})$, $(|\hat{\text{UA}}|_{al}, |\hat{\text{UE}}|_{al}, |\hat{\text{UA}}'|_{al}) \in Call$;*
3. *if $p \equiv \hat{\mathcal{I}}(pr, \hat{\boldsymbol{d}}) \rightsquigarrow^+ \hat{\varsigma}$ such that $\hat{\varsigma}$ is a final state, then $|\hat{\varsigma}|_{al} \in Final$.*

The proof of this theorem is the same as that of CFA2 modulo our definitions of corresponding entry and "same-level" states (and our omission of local environments from stack frames). The key step is ensuring that each called continuation is properly identified on the stack. In CFA2, where only a single continuation is possible, the continuation resides at the penultimate stack frame. In MCCFA2, the continuation could reside arbitrarily-deep in the stack. We address this possibility by connecting the path structure induced by corresponding entries and "same-level" states to stack behavior. The proof appears in a technical report [5].

Summarization Completeness. Completeness is the property that only abstract paths p initiated by $\hat{\mathcal{I}}(pr, \hat{\boldsymbol{d}})$ are contained in the results of summarization. Formally, we have the following:

Theorem 3 (Completeness).
After summarization,

1. *if* $(\widetilde{\text{UA}}, \widetilde{\text{CEE}}, n) \in$ *Summary then there exists* $p \equiv \hat{\mathcal{I}}(pr, \hat{\boldsymbol{d}}) \rightsquigarrow^* \widehat{\text{UA}} \rightsquigarrow^+ \widehat{\text{CEE}}$ *such that* $\widetilde{\text{UA}} = |\widehat{\text{UA}}|_{al}$*,* $\widetilde{\text{CEE}} = |\widehat{\text{CEE}}|_{al}$*, and* $\widehat{\text{UA}} \equiv_p \widehat{\text{CEE}}$ *by* n*; and*
2. *if* $(\widetilde{\text{UA}}, \widetilde{\text{UE}}, \widetilde{\text{UA}}') \in$ *Call, then there exists* $p \equiv \hat{\mathcal{I}}(pr, \hat{\boldsymbol{d}}) \rightsquigarrow^* \widehat{\text{UA}} \rightsquigarrow^+ \widehat{\text{UE}} \rightsquigarrow \widehat{\text{UA}}'$ *such that* $\widetilde{\text{UA}} = |\widehat{\text{UA}}|_{al}$*,* $\widetilde{\text{UE}} = |\widehat{\text{UE}}|_{al}$*,* $\widetilde{\text{UA}}' = |\widehat{\text{UA}}'|_{al}$*, and* $\widehat{\text{UA}} = CE_p(\widehat{\text{UE}})$*;*
3. *if* $\tilde{\varsigma} \in$ *Final, then there exists* $p \equiv \hat{\mathcal{I}}(pr, \hat{\boldsymbol{d}}) \rightsquigarrow^+ \hat{\varsigma}$ *such that* $\tilde{\varsigma} = |\hat{\varsigma}|_{al}$ *and* $\hat{\varsigma}$ *is a final state.*

The proof of this theorem strongly resembles the corresponding proof for CFA2 except for the use of summaries to extend paths; it appears in a technical report [5].

A CFA2-produced summary (*entry*, *exit*) records not only that *exit* is reachable from *entry* but also such that the intervening evaluation perfectly balances proper calls and returns. The path segments represented by these summaries exhibit the property of *stack irrelevance*, that is, that the evaluation of these path segments is not influenced by nor influences the stack of the *entry* state. This property allows abstract paths to be synthesized by replacing irrelevant suffixes of the stack.

When multiple continuations are present, user and continuation calls can pop arbitrary portions of the stack, even below the stack of the *entry* state. Hence, a summary (*entry*, *exit*, n) subject to the same call–return balance restriction does not enjoy this property. However, such paths can be *normalized*, removing irrelevant suffixes of each constituent invocation, so that summaries can be employed in the same way. This ability is critical to demonstrating completeness, one of our technical contributions.

9 Multi-return λ-Calculus

Shivers and Fisher [16] introduce the multi-return λ-calculus (MRLC) as an extension of a direct-style λ-calculus in which return points become an explicit (though second-class) language construct. With this mechanism, MRLC essentially provides user-level access to multiple escape continuations without the severe notational overhead of CPS. This access makes MRLC adept at expressing programs from particular control-heavy domains such as LR parsing, backtracking search, and functional tree transformations [20]. MRLC is designed to translate into MCPS so our analysis framework is keenly poised to handle these domains as well.

Shivers and Fisher illustrate the utility of MRLC with a parsimonious `filter` program which employs multiple return points to reuse as much of the input list as possible. We consider MCCFA2 applied to the MCPS transformation of this program:

```
(λ0 (? ws k0)
  (define1 (recur xs k1 k2)
    (case xs k1
            (λ3 (y ys)
              (? y (λ5 (t)
                     (if t
                       (λ7 () (recur ys k1 (λ9 (zs) (cons y zs k2)10))8)
                       (λ11 () (recur ys (λ13 () (k2 ys)14) k2)12))6))4))2)
  (recur ws (λ16 () (k0 ws)17) k0)15)
```

The primitive `case` procedure performs case analysis on its first argument, deconstructs it, and invokes one of its continuations on the subparts. To exercise the full behavior of this program, we apply it to $\langle$<**havoc**>, $\top_{list}\rangle$ where <**havoc**> is an arbitrary primitive predicate.

The table in Fig. 10 presents the destination and content of each analysis fact MCCFA2 discovers, in a possible order of discovery. Each call is a triple consisting of the calling procedure entry, the call site, and the called procedure entry. Primitive procedures have opaque representations of the form <*name*>. Each summary is a triple consisting of a procedure entry, procedure exit, and continuation index. A procedure exit is merely the exit site with the result values implicit. Primitive procedure exit sites are not represented in the program so we denote them with a pair (<*name*>, n) of primitive identifier and continuation index, The final state is simply the program result value.

1	CALL	(λ_0,$call_{15}$,λ_1)	11	SUMMARY	(<if>,(<if>,1),1)
2	CALL	(λ_1,$call_2$,<case>)	12	CALL	(λ_1,$call_8$,λ_1)
3	SUMMARY	(<case>,(<case>,1),1)	13	SUMMARY	(<if>,(<if>,2),2)
4	SUMMARY	(λ_1,(<case>,1),1)	14	CALL	(λ_1,$call_{12}$,λ_1)
5	SUMMARY	(λ_0,$exit_{17}$,1)	15	SUMMARY	(λ_1,$exit_{14}$,2)
6	FINAL	($\top_{list}$)	16	CALL	(λ_1,$call_{10}$,<cons>)
7	SUMMARY	(<case>,(<case>,2),2)	17	SUMMARY	(λ_0,$exit_{14}$,2)
8	CALL	(λ_1,$call_4$,<?>)	18	SUMMARY	(<cons>,(<cons>,1),1)
9	SUMMARY	(<?>,(<?>,1),1)	19	SUMMARY	(λ_1,(<cons>,1),2)
10	CALL	(λ_1,$call_6$,<if>)	20	SUMMARY	(λ_0,(<cons>,1),1)

Fig. 10. An MCCFA2 analysis

10 Continuation Age Analysis

When a user call is made with multiple continuations, Might and Shivers' generalization of Orbit's stack-management policy [14] dictates that all dead frames are popped from the stack before the target procedure is entered. Dead frames are typically determined dynamically by comparing the frame pointers of the call's continuations (as seen in the *UEval* rule of our concrete semantics) which requires an MCPS-based compiler to emit comparison code at each call site.

Vardoulakis and Shivers [20] introduced *continuation age* analysis which attempts to statically determine the relative ages among each call's continuations. They build their analysis into a pre-existing finite-state k-CFA [15,18] analysis framework. We can perform continuation age analysis directly on the pushdown model MCCFA2 constructs without modifying the MCCFA2 implementation.

For each call site $ucall = (f\, \boldsymbol{e}\, \boldsymbol{k}^+)_\gamma$ where $|\boldsymbol{k}| > 1$, let

$$\phi_0 = \{(\widetilde{\text{UA}}_0, \widetilde{\text{UE}}_0, \boldsymbol{k}) : (\widetilde{\text{UA}}_0, \widetilde{\text{UE}}_0, \widetilde{\text{UA}}) \in \mathit{Call}, \widetilde{\text{UE}}_0 = (\mathit{ucall}, h) \text{ for some } h\}.$$

That is, ϕ_0 is a set of triples where each triple contains a $\widetilde{\mathit{UEval}}$ state focused on *ucall*, its corresponding entry, and its continuation argument vector (comprising only continuation references). Find the fixed point of $f(\phi_0)$ defined as

$$\begin{aligned} f(\phi_0)(\phi) = \phi_0 \cup \{(\widetilde{\text{UA}}_0, \widetilde{\text{UE}}_0, \boldsymbol{q}') : \; & (\widetilde{\text{UA}}, \widetilde{\text{UE}}, \boldsymbol{q}) \in \phi, \\ & \boldsymbol{q} \in \mathit{CVar}^+, \\ & (\widetilde{\text{UA}}_0, \widetilde{\text{UE}}_0, \widetilde{\text{UA}}) \in \mathit{Call}, \\ & \boldsymbol{k} = \mathit{CP}(\mathit{ulam}) \text{ where } \widetilde{\text{UA}} = (\mathit{ulam}, \hat{\boldsymbol{d}}, h) \text{ for some } \hat{\boldsymbol{d}} \text{ and } h, \\ & \boldsymbol{q}_0 = \mathit{LC}(\mathit{ucall}) \text{ where } \widetilde{\text{UE}}_0 = (\mathit{ucall}, h) \text{ for some } h, \\ & \boldsymbol{q}' = [\boldsymbol{k} \mapsto \boldsymbol{q}_0](\boldsymbol{q}) \,\}. \end{aligned}$$

The function f considers all triples in its argument ϕ that have a continuation argument vector $\boldsymbol{q}$ consisting solely of continuation references. The callers of each such triple's entry state are used to construct new triples containing that caller, its corresponding entry, and $\boldsymbol{q}$ mapped over $[\boldsymbol{k} \mapsto \boldsymbol{q}_0]$ which permutes the outer caller's continuation vector to match the inner caller's.

By MCCFA2 soundness, this process will accumulate all continuation argument vectors that contain some *clam* that is eventually bound to a reference at the original *ucall*. Given a fixed point ϕ of $f(\phi_0)$, we can consider only the continuation vectors that contain a *clam*. One of the many ways to use the resultant vectors is to map each to the set of indices at which a *clam* is found. Any indices in the intersection of these sets are those of the youngest continuation. If the intersection is empty, the union contains indices that *may* be the youngest. This information may decrease the number of comparisons necessary to determine the youngest at run time.

11 Related Work

There are many instances of pushdown control-flow analysis for higher-order languages [3,6,8,19]. This work is framed around CFA2. In their extension of CFA2 to handle first-class control [21], Vardoulakis and Shivers outline two approaches to extend CFA2 to support exceptions without sacrificing precision:

1. Outlined in [21, Sect. 2.4], they propose to let exit points encapsulate a pair of values, the first representing the result of standard control flow and the second of exceptional control flow. Since procedures don't exit naturally and

exceptionally simultaneously, this pair behaves as a sum with the position of the value providing an additional bit of information. Our approach generalizes this approach in a sense by providing as many bits as the continuation position takes to encode.

2. Outlined in [21, Sect. 5.5], they propose translating the program into 2CPS, using the first continuation for standard control flow and the second for exceptional—the standard "double-barrelled" CPS. In this approach, two distinct summary relations must be maintained by the summarization algorithm "to not confuse exceptional with ordinary control flow". As in the previous approach, the caller syntax is inspected to determine whether it can handle the type of return the summary represents, this time looking for a literal λ term in the appropriate continuation position. Our approach extends this approach in the obvious way, generalizing to arbitrarily many continuations and using indices to distinguish summaries. This generalization is not free, however, as we have made significant modifications to the summarization algorithm and soundness/completeness arguments, in turn.

Pushdown exception-flow analysis has also been applied in the context of object-oriented programs [12]. Like extended CFA2, the treatment is specialized to exceptions and not the multiple continuations in general.

Acknowledgments. This material is partially based on research sponsored by DARPA under agreement number AFRL FA8750-15-2-0092 and by NSF under CAREER grant 1350344. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Adams, N., Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J.: ORBIT: an optimizing compiler for scheme. In: SIGPLAN 1986. ACM, New York (1986)
2. Appel, A.W.: Compiling with Continuations. Cambridge University Press, Cambridge (2007)
3. Earl, C., Might, M., Van Horn, D.: Pushdown control-flow analysis of higher-order programs. In: Workshop on Scheme and Functional Programming (2010)
4. Felleisen, M.: On the expressive power of programming languages. Sci. Comput. Program. **17**(1), 35–75 (1991)
5. Germane, K., Might, M.: Multi-continuation pushdown analysis. Technical report, January 2019. http://kimball.germane.net/germane-mccfa2-techreport.pdf
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 Symposium on Principles of Programming Languages. POPL 2016, pp. 691–704. ACM, New York (2016)
7. Hieb, R., Dybvig, R.K., Bruggeman, C.: Representing control in the presence of first-class continuations. In: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation. PLDI 1990, pp. 66–77. ACM, New York (1990)

8. Johnson, J.I., Van Horn, D.: Abstracting abstract control. In: Proceedings of the 10th ACM Symposium on Dynamic languages, pp. 11–22. ACM (2014)
9. Kennedy, A.: Compiling with continuations, continued. In: Proceedings of the 12th ACM International Conference on Functional Programming. ICFP 2007, pp. 177–190. ACM, New York (2007)
10. Kim, J., Yi, K., Danvy, O.: Assessing the overhead of ML exceptions by selective CPS transformation, vol. 5, January 1998
11. Ley-Wild, R., Fluet, M., Acar, U.A.: Compiling self-adjusting programs with continuations. In: Proceedings of the 13th ACM International Conference on Functional Programming. ICFP 2008, pp. 321–334. ACM, New York (2008)
12. Liang, S., Sun, W., Might, M., Keep, A., Horn, D.V.: Pruning, pushdown exception-flow analysis. In: Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pp. 265–274. IEEE Computer Society (2014)
13. Might, M.: Environment analysis of higher-order languages (2007)
14. Might, M., Shivers, O.: Environment analysis via ΔCFA. In: Conference Record of the 33rd ACM Symposium on Principles of Programming Languages. POPL 2006, pp. 127–140. ACM, New York (2006)
15. Shivers, O.: Control-flow analysis of higher-order languages. Ph.D. thesis. Carnegie Mellon University (1991)
16. Shivers, O., Fisher, D.: Multi-return function call. J. Funct. Program. **16**(4), 547–582 (2006)
17. Thielecke, H.: Comparing control constructs by double-barrelled CPS. Higher-Order Symb. Comput. **15**(2), 141–160 (2002)
18. Van Horn, D., Might, M.: Abstracting abstract machines. In: Proceedings of the 15th ACM International Conference on Functional Programming. ICFP 2010, pp. 51–62. ACM, New York (2010)
19. Vardoulakis, D., Shivers, O.: CFA2: a context-free approach to control-flow analysis. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 570–589. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_30
20. Vardoulakis, D., Shivers, O.: Ordering multiple continuations on the stack. In: Proceedings of the 20th ACM Workshop on Partial Evaluation and Program Manipulation. PEPM 2011, pp. 13–22. ACM, New York (2011)
21. Vardoulakis, D., Shivers, O.: Pushdown flow analysis of first-class control. In: Proceedings of the 16th ACM International Conference on Functional Programming. ICFP 2011, pp. 69–80. ACM, New York (2011)