A Posteriori Environment Analysis with Pushdown Delta CFA

Kimball Germane   Matthew Might
University of Utah, USA

Abstract

Flow-driven higher-order inlining is blocked by free variables, yet current theories of environment analysis cannot reliably cope with multiply-bound variables. One of these, ΔCFA, is a promising theory based on stack change but is undermined by its finite-state model of the stack. We present Pushdown ΔCFA which takes a ΔCFA-approach to pushdown models of control flow and can cope with multiply-bound variables, even in the face of recursion.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors and Optimization

Keywords Static analysis; Environment analysis

1. Introduction

Higher-order procedure inlining requires higher-order considerations. To illustrate, consider the Scheme program

```
(let ([f (λ (x h) (if (zero? x) (h) (λ () x)))])
  (f 0 (f 3 #f)))
```

which first binds `f` and then invokes it twice, the second time with its own result.

It seems apparent that `(λ () x)` can be inlined at `(h)`: only closures over `(λ () x)` flow to `h` at `(h)` and `x` is in scope at `(h)`. However, `x`'s *binding* in the environment of `(h)` is different than its binding in the environment of `(λ () x)` when the closure that flows to `h` is captured. Were we to proceed with the inlining, the meaning of the program would change from 3 to 0. Consequently, this inlining is unsafe.

Shivers [13, Ch. 10] outlines three sufficient safety conditions for inlining the operator f of a call site $call$. These are:

1. That closures over only a single λ-term lam flow to f, i.e., that the inlining operation is even sensible. This condition concerns control flow and can be established by a control-flow analysis such as k-CFA [12, 13] or CFA2 [16].
2. That each free variable x of lam is in scope at $call$. The inlining operation syntactically replaces f with lam in $call$ so each free variable x of lam must be bound in the environment of $call$. This condition is trivial to establish under lexical binding.
3. That the binding of each free variable x of lam is the same in the environment of $call$ as it is in the environment of lam *before* inlining. This condition concerns environments and may be established by an *environment analysis*.

Environment analysis plays a critical role: the potential inline in the previous example met the first two conditions but not the third.

Several theories of environment analysis have been developed. To see them in action, we consider a higher-order program with a non-trivial nested loop, seen in Figure 1.[1] This program recurs on a list of numbers and produces a list of accumulated sums. For instance, when invoked on `(list 1 2 3 4 5)` (the first five natural numbers), it produces `(list 1 3 6 10 15)` (the first five triangle numbers). At each step, the number at the front of the list is captured in a closure to `map` across the rest of the list. Before doing so, however, `tri` recurs on the rest of the list, so the list of accumulated sums is built incrementally from back to front.

Control-flow analysis reveals that each operator of the call site `(f w)` is a closure over `(λ (u) (+ y u))` and that its free variable `y` is in scope there, satisfying the first two inlining conditions. As the previous example demonstrates, the bindings of `y` in the environment of `(λ (u) (+ y u))` and the environment of `(f w)` may not agree. To establish inlining safety, we turn to environment analysis.

We now look at the approaches taken by five existing environment analyses and examine how each behaves on this example, in hopes to expose the issues at hand and give a taste of possible approaches. Each of these analyses is built upon abstract interpretation [3].

Reflow analysis [13, Ch. 8] re-runs the control-flow analysis—starting it mid-stream—for each binding of closure-captured variables, isolating a single binding each time. When applied to the binding of `y` in this example, reflow analysis successfully proves the inlining safe at the cost of another control-flow analysis.

ΓCFA [11] keeps track of the number of concrete counterparts to each abstract binding, using abstract garbage collection to reap stale bindings. Applied to this example, it attempts to prove that only a single binding of `y` may be live at any point. As multiple bindings of `y` are co-live within nested recursive calls, it fails to prove the inlining safe.

Binding anodization [9] attempts to generalize reflow analysis by isolating bindings according to a user-specified policy. Unlike reflow analysis, however, it is performed in conjunction with and not after the initial control-flow analysis. Specific binding isolation policies are tailored to specific binding patterns and, though programs typically exhibit myriad binding patterns, only a single policy may be active for each analysis. In short, there is likely a policy which justifies this inlining, but the user

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

POPL'17, January 15–21, 2017, Paris, France
ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009899

[1] This program exhibits an unspecialized use of `map` which is unlikely to arise in practice but illustrates a particular issue that blocks most environment analyses.

```
(λ (us)
  (letrec ([tri (λ (xs)
                 (match xs
                   [(list) (list)]
                   [(cons y ys) (letrec ([map (λ (f zs)
                                               (match zs
                                                 [(list) (list)]
                                                 [(cons w ws) (cons (f w) (map f ws))]))])
                                  (cons y (map (λ (u) (+ y u)) (tri ys))))]))])
    (tri us)))
```

Figure 1. Accumulating sum program

must manually devise it for this particular binding pattern and implement it up front.

UVA [2] casts binding equivalence in terms of graph reachability: using a control-flow graph built from the control-flow analysis, it equivocates the binding and use of a variable that are graph-connected but only if no connecting path touches a rebinding of that variable. As y is recursively rebound between a particular binding and use in this example, UVA fails to justify this inlining.

ΔCFA [10] uses stack behavior as a proxy for the environment and identifies bindings allocated at the same point in the stack history. However, its finite-state abstraction of the stack is coarse and the recursion leads it to conflate distinct bindings of y. Consequently, it too is unable to justify this inlining.

In summary, only reflow analysis is able to justify this inlining (but lacks formal justfication for its correctness).

The ΔCFA approach is worth revisiting. In fact, the failure of ΔCFA to justify this inlining is not inherent in its approach but is instead a result of the imprecise finite-state abstraction afforded by its host analysis, k-CFA [12, 13].

1.1 A Brief ΔCFA Primer

ΔCFA comprises semantic machinery, an environment theory, and a sound abstraction of the semantic machinery with respect to that theory.

ΔCFA meticulously tracks stack change during program evaluation in terms of *frame strings*, sequences of stack actions. Having done so, ΔCFA can reconstruct a frame string representation of the intervening stack motion between any two points in evaluation.

The environment theory of ΔCFA leverages this ability by imagining that all environment bindings are allocated on the stack (at least initially) and viewing a stack action as the introduction or removal of particular bindings. It can then reason about binding equivalence by comparing frame strings.

In the exact programming language semantics, where perfect frame string reconstruction is possible, these comparisons yield exact though incomputable answers. The finite-state approximation employed by the abstract semantics of ΔCFA brings it into the realm of computability but out of the realm of recursion.

1.2 Pushdown ΔCFA

Pushdown models of control flow provide a fundamentally more precise account of the stack than their finite-state forebears. This feature makes them a natural foundation for ΔCFA's stack-centric environment theory. Moreover, stack behavior is embedded in the pushdown model itself meaning that it alone is sufficient to apply ΔCFA's environment theory to answer binding equivalence questions. This feature provides a mechanism to perform environment analysis after the model is constructed, i.e., after control-flow analysis, and allows us to avoid coupling the implementations and correctness arguments of the underlying flow analysis with those of ΔCFA.

$$
\begin{array}{rcl}
u \in \mathit{UVar} & = & \text{a set of identifiers} \\
k \in \mathit{CVar} & = & \text{a set of identifiers} \\
\mathit{lam} \in \mathit{Lam} & = & \mathit{ULam} + \mathit{CLam} \\
\mathit{ulam} \in \mathit{ULam} & ::= & (\lambda_\ell\, (u^*\, k)\, \mathit{call}) \\
\mathit{clam} \in \mathit{CLam} & ::= & (\lambda_\gamma\, (u^*)\, \mathit{call}) \\
\mathit{call} \in \mathit{Call} & = & \mathit{UCall} + \mathit{CCall} \\
\mathit{ucall} \in \mathit{UCall} & ::= & (f\, e^*\, q)_\ell \\
\mathit{ccall} \in \mathit{CCall} & ::= & (q\, e^*)_\gamma \\
e, f \in \mathit{UExp} & = & \mathit{UVar} + \mathit{ULam} \\
q \in \mathit{CExp} & = & \mathit{CVar} + \mathit{CLam} \\
\psi \in \mathit{Lab} & = & \mathit{ULab} + \mathit{CLab} \\
\ell \in \mathit{ULab} & = & \text{a set of labels} \\
\gamma \in \mathit{CLab} & = & \text{a set of labels} \\
\mathit{pr} \in \mathit{Pr} & = & \{\mathit{ulam} : \mathit{ulam} \in \mathit{ULam}, \mathrm{closed}(\mathit{ulam})\}
\end{array}
$$

Figure 2. Partitioned CPS λ-calculus syntax

In this paper, we present Pushdown ΔCFA, an a posteriori environment analysis that takes just such an approach. We employ the control-flow analysis CFA2 [16] to host ΔCFA, due to their common heritage. We briefly introduce their shared language, the CPS λ-calculus, in Section 2 and its semantics in Section 3. We review the essential aspects of CFA2 in Section 5 and ΔCFA in Section 6. We introduce Pushdown ΔCFA proper in Section 7 and walk through an example in Section 8. Finally, we discuss related and future work in Section 9.

Readers already familiar with ΔCFA or CFA2 may want to skim its respective section to become acquainted with our notation.

2. Partitioned CPS λ-Calculus

ΔCFA and CFA2 both operate over a continuation-passing style (CPS) λ-calculus. This language is best seen as an intermediate representation produced from a direct-style source program. CPS is convenient for us in part because it allows us to distinguish source-program tail calls, inner calls, and returns solely by the evaluation control string (see Section 6.3). We assume the CPS translation keeps terms present in the original program (*user-world* terms) distinct from terms it introduces (*continuation-world* terms).

Figure 2 presents the syntax of this CPS language. Within the user and continuation world, respectively, there are the typical λ-calculus syntax classes: variables references u and k, λ-terms ulam and clam, and calls ucall and ccall. We assume that both λ-terms

$$
\begin{aligned}
\varsigma \in \mathit{State} &= \mathit{Eval} + \mathit{Apply} \\
\text{E} \in \mathit{Eval} &= \mathit{Call} \times \mathit{BEnv} \times \mathit{VEnv} \times \mathit{Time} \\
\text{A} \in \mathit{Apply} &= \mathit{Proc} \times D^* \times \mathit{VEnv} \times \mathit{Time} \\
\beta \in \mathit{BEnv} &= \mathit{Var} \rightharpoonup \mathit{Time} \\
\mathit{ve} \in \mathit{VEnv} &= \mathit{Var} \times \mathit{Time} \rightharpoonup D \\
t \in \mathit{Time} &= \text{a countably-infinite ordered set} \\
d, c \in D &= \mathit{Proc} \\
\mathit{proc} \in \mathit{Proc} &= \mathit{Clos} + \{\mathit{halt}\} \\
\mathit{clos} \in \mathit{Clos} &= \mathit{Lam} \times \mathit{BEnv}
\end{aligned}
$$

Figure 3. Standard state space

and calls are given a distinct label drawn from their native world and let $L_{pr}(\psi)$ denote the ψ-labelled term in program pr. (Most of the time, the program pr is apparent from context and, in such cases, we omit the subscript.) A program pr is simply a closed ulam.

Given a user-call ucall with label ℓ, we let $OE_{pr}(\ell)$ denote ucall's operator expression f. Similarly, given a (user- or continuation-world) call call with label ψ, we let $IE_{pr}(\psi, i)$ denote the ith argument expression e of call. We will sometimes abuse these notations by providing the call syntax itself and, later, its enclosing machine state, but the intention should always be clear from context.

Given a λ-term lam with label ψ, we let $BP_{pr}(\psi, i)$ denote the ith parameter u of lam. BP_{pr}, coupled with IE_{pr}, will allow us to track value flow across calls. With that same label ψ, we let $B_{pr}(\psi)$ denote the set of parameters $\{u_1, \ldots, u_n\}$. Given a call site call with label ψ, we let $\mathit{Scope}_{pr}(\psi)$ denote the set of user variables $\{u_1, \ldots, u_n\}$ in lexical scope at call.

We impose a restriction on programs that a continuation variable k cannot appear free under a ulam. This restriction prevents continuations from escaping (and prohibits first-class continuations, in turn) but ensures that the dynamic continuations can be managed by a stack. We discuss extending our analysis to remove this restriction in Section 9.

3. Standard Semantics

We define the semantics of our CPS λ-calculus in terms of a small-step abstract machine; Figure 3 presents its state space. The machine operates over states $\varsigma \in \mathit{State}$ with computation alternating between the atomic evaluation of an operator and its arguments, performed from states $\text{E} \in \mathit{Eval}$, and the application of that operator to those arguments, performed from states $\text{A} \in \mathit{Apply}$.

Each state ς has a value environment ve, which serves as a store, and timestamp t. Each Eval state E has a call call closed over by a binding environment β. Each Apply state A has a procedure proc to be applied to the argument vector $\mathbf{d}$.

Figure 4 presents the Eval and Apply transitions of the machine. In the Eval transition, the operator and arguments to a call are atomically evaluated using $\mathcal{A}$ and the arguments are packaged in a vector for the subsequent transition. In the Apply transition, the environment β of the operator is extended with bindings for the arguments. This binding takes a mapping from a parameter variable x to the timestamp t' of the successor state. The value environment ve is likewise extended, mapping the bindings themselves to the argument values.

An Eval transition from a user-world call and an Apply transition of a user-world procedure will evaluate and bind a continuation argument, respectively, reflected by $q^?$ and $c^?$. These transitions respect the user- and continuation-world distinction: values derived from user-world terms are bound only to user-world variables and values derived from continuation-world terms (continuations) are bound only to continuation-world variables.

Eval

$$\overbrace{((f\, e_1 \ldots e_n\, q^?)_\psi, \beta, \mathit{ve}, t)}^{\varsigma} \to (\mathit{proc}, \langle d_1, \ldots, d_n, c^? \rangle, \mathit{ve}, t')$$

$$
\begin{aligned}
t' &= \mathit{tick}(\varsigma) \\
\mathit{proc} &= \mathcal{A}(f, \beta, \mathit{ve}) \\
d_i &= \mathcal{A}(e_i, \beta, \mathit{ve}) \text{ for } i = 1, \ldots, n \\
c &= \mathcal{A}(q, \beta, \mathit{ve})
\end{aligned}
$$

Apply

$$\overbrace{(\mathit{proc}, \langle d_1, \ldots, d_n, c^? \rangle, \mathit{ve}, t)}^{\varsigma} \to (\mathit{call}, \beta', \mathit{ve}', t')$$

$$
\begin{aligned}
\mathit{proc} &= ((\lambda_\psi\, (u_1 \ldots u_n\, k^?)\, \mathit{call}), \beta) \\
t' &= \mathit{tick}(\varsigma) \\
\beta' &= \beta[u_1 \mapsto t', \ldots, u_n \mapsto t', k \mapsto t'] \\
\mathit{ve}' &= \mathit{ve}[(u_1, t') \mapsto d_1, \ldots, (u_n, t') \mapsto d_n, (k, t') \mapsto c]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{A}(x, \beta, \mathit{ve}) &= \mathit{ve}(x, \beta(x)) \\
\mathcal{A}(\mathit{lam}, \beta, \mathit{ve}) &= (\mathit{lam}, \beta)
\end{aligned}
$$

$$\mathcal{I}(pr, \langle d_1, \ldots, d_n \rangle) = ((pr, \emptyset_\beta), \langle d_1, \ldots, d_n, \mathit{halt} \rangle, \emptyset_{\mathit{ve}}, t_0)$$

Figure 4. Standard Semantics

Each transition uses tick to derive the timestamp t' of the successor state from ς. Depending on the application, different instantiations of tick are useful. For our purposes, the most important feature of an instantiation is that a state's timestamp is fresh with respect to its predecessors'. Freshness guarantees that a time found within a state can be treated as a reference to the state it stamps. We use this property to translate time-oriented statements in ΔCFA's environment theory to state-oriented ones (see Section 7). The trivial instantiation $\mathit{Time} = \mathbb{N}$ and $\mathit{tick}(\varsigma) = t_\varsigma + 1$ exhibits this feature.

The atomic evaluator function $\mathcal{A}$ takes an atomic expression—a variable x or a λ-term lam—along with a binding environment β and value environment ve, and produces a value. If the atomic expression is a variable x, its binding is looked up in β which is then used to key into ve. If it's a lam, it is combined with β to produce a closure.

The injection function $\mathcal{I}$ injects a program pr and its arguments $\mathbf{d}$ into an Apply state with an empty value environment and an epoch time (here 0).

4. An Example Program

When reviewing CFA2 and ΔCFA, we will use the factorial program of Figure 5 which presents its direct-style expression and CPS translation. User-world terms are labelled with an uppercase letter while continuation-world terms with a lowercase.

In the direct-style program, we use recursive procedures via `letrec`, integers, the primitive procedures `*` and `-` over the inte-

```
(λ_A (m)
  (letrec ([fact (λ_B (n)
                   (if0 n
                     1
                     (* n (fact (- n 1)))))])
    (fact m)))
```

```
(λ_A (m k0)
  (letrec ([fact (λ_B (n k1)
                   (if0 n
                        (λ_a () (k1 1))
                        (λ_b ()
                          (- n 1
                             (λ_c (n-1)
                               (fact n-1
                                     (λ_d (a)
                                       (* n a k1))))))))])
    (fact m k0)))
```

Figure 5. The fact program and its CPS translation

gers, and the branching construct if0. We have left the letrec construct in the CPS translation for clarity, but we treat (letrec ([f *lam*]) *call*) as (let ([f (Y (λ (f) *lam*))]) *call*) where Y is a call-by-value fixed-point combinator. Additionally, we have expressed the if0 form as a primitive procedure that accepts *two* nullary continuations in addition to its argument n. This rendering promotes uniformity in our explanations but doesn't introduce any significant issues to our analysis.

5. CFA2

CFA2 [16] is a higher-order flow analysis which precisely matches returns to their corresponding calls. CFA2 achieves this by modelling program evaluation as a pushdown automaton in which the stack records return points.

5.1 Evaluation Paths

An *evaluation path* P is some contiguous sequence of machine steps $\varsigma_1 \rightarrow \varsigma_2 \rightarrow \ldots \rightarrow \varsigma_n$, often headed by the initial machine state $\mathcal{I}(pr, \mathbf{d})$. Paths are often interjected with other relations such as the reflexive $\rightarrow^0$, the transitive closure $\rightarrow^+$ over $\rightarrow$, and the reflexive, transitive closure $\rightarrow^*$.

5.2 Path Decomposition

Program evaluation produces paths with rich structure, arising from both the steady alternation between *Eval* and *Apply* states and the restrictive continuation protocol to which procedures adhere.

To illustrate, consider the fact program of Figure 5. If we apply this program to 0, we obtain the evaluation path depicted in Figure 6. Each node in this depiction represents a state which is labelled according to that state's type and annotated with its timestamp and, where applicable, its operator label. Type labels are a sequence of three characters and can be decoded as follows:

1. The first character of U or C signifies which of the user world *User* and continuation world *Cont*, respectively, has control at that state. For *Eval* states, this is the world of the call; for *Apply* states, this is the world of the applied procedure.
2. The second character of E or A signifies whether it is an *Eval* or *Apply* state, respectively.
3. The third character qualifies the first two characters. For an *Eval* state, an I signifies an *inner* state. For a *Cont-Apply* state, an I signifies that its predecessor state (an *Eval* state) is an inner state. An E can suffix an *Eval* state to signify that it is an *exit* state and an R can suffix a *Cont-Apply* state to signify that its predecessor is an exit state. A *User-Apply* state has only one variant, so it brandishes the two-character label UA.

Labels convey a surprising amount of information about the role its state plays in evaluation. For example, a *User-Eval-Exit* label UEE signifies a tail call and a *Cont-Apply-Return* label CAR signifies a return to a particular point in a procedure body. Remarkably, the appropriate label for a state—that state's *classification*—can be determined entirely by the shape of the state's control syntax, or the shape of its predecessor's.

The following table illustrates this correspondence.

Label	Syntax shape	Label	Syntax shape
UEE	$(f\ e^*\ k)$	UA	$(\lambda_\ell\ (u^*\ k)\ call)$
UEI	$(f\ e^*\ clam)$	UA	$(\lambda_\ell\ (u^*\ k)\ call)$
CEE	$(k\ e^*)$	CAR	$(\lambda_\gamma\ (u^*)\ call)$
CEI	$(clam\ e^*)$	CAI	$(\lambda_\gamma\ (u^*)\ call)$

Eval state variants are on the left half of the table with their successor *Apply* state variants on the right. *Eval* state classification as "exit" (E) or "inner" (I) depends merely on whether its call's continuation expression is k or $clam$, respectively. A *User-Apply* state is always a procedure entry state; *Cont-Apply* state classification as "return" (R) or "inner" (I) is dictated by the state's predecessor.

CFA2 emphasizes the roles signified by these labels by *decomposing* paths according to them. Figure 7 presents a depiction of the same evaluation path of Figure 6, decomposed. In this depiction, nodes are colored according to the *invocation* to which they belong. An invocation is a subsequence of an evaluation path which begins with a procedure entry state (labelled UA) and includes every state evaluating that procedure's body in an extension of the entry state's environment.[2] The vertical position of nodes in this depiction corresponds to the height of the stack in the denoted state. When the tail call of the white UEE node is made, neither a return point to nor the environment of the exiting procedure needs to be saved, so the stack doesn't grow. In contrast, when the *inner* call of the yellow UEI node is made, a frame is pushed to preserve the calling procedure's environment and return point.

To increase our intuition about decomposition, let's consider Figure 8 which presents the decomposed evaluation path of the factorial program applied to 1. We can see that, even without program syntax, a decomposed path provides a fairly clear picture of the structure of evaluation. Notice that the inner invocation of fact applied to 0 has precisely the same form as that of the path in Figure 7. In fact, modulo the continuation, the machine states are precisely the same as well.

Our informal definition of "invocation" appeals directly to the environments of states. As we will see, CFA2 sheds environments as we know them, so we must find some other means of defining it. Fortunately, it is no coincidence that states in the same invocation have the same stack height and CFA2 defines the complementary notions of a *corresponding entry* and *same-context entry* to make this connection. Together, these notions inductively decompose a path according to the classification of the path's states and we formally define "invocation" in terms of this decomposition.

[2] To be precise, evaluation must be in an extension of the entry state's *successor's* environment, after the arguments and continuation are bound.

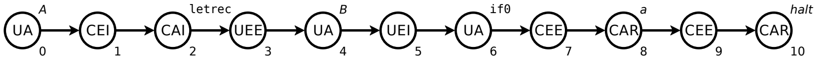

Figure 6. Evaluation path of $\mathrm{fact}(0)$

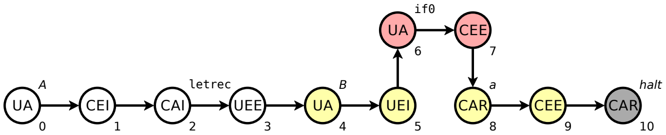

Figure 7. Decomposed evaluation path of $\mathrm{fact}(0)$

5.2.1 A Note on Notation

Just as we've let E and A stand for *Eval* and *Apply* states, we will now let UA and UEI, for example, stand for states labelled UA and UEI. We will also synthesize corresponding domains, so that, for example, $\mathrm{UA} \in \mathit{UserApply}$ and $\mathrm{UEI} \in \mathit{UserEvalInner}$. Finally, we will omit characters when irrelevant: for example, we let EE stand for an exit state, user- or continuation-world, and UE stand for a user call, inner or tail.

5.3 Path Decomposition, Formally

We first review CFA2's definitions of *corresponding entry* and *same-context entry*, adapting them to our notation.

Definition 1 (Corresponding and Same-Context Entry, [16]). *Let $CE(\varsigma)$ denote the* corresponding entry *of a state ς in path P. For path $P \equiv \mathrm{UA} \rightarrow^* \varsigma$, $CE(\varsigma) = \mathrm{UA}$ if:*

1. $P \equiv \mathrm{UA} \rightarrow^0 \varsigma$;
2. $P \equiv \mathrm{UA} \rightarrow^* \varsigma' \rightarrow \varsigma$, $\mathrm{UA} = CE(\varsigma')$, $\varsigma' \notin \mathit{UserEval}$, *and* $\varsigma' \notin \mathit{ContEvalExit}$; *or*
3. $P \equiv \mathrm{UA} \rightarrow^+ \mathrm{UEI} \rightarrow \mathrm{UA}_0 \rightarrow^+ \mathrm{CEE} \rightarrow \varsigma$, $\mathrm{UA} = CE(\mathrm{UEI})$, *and* $\mathrm{UA}_0 \in CE^*(\mathrm{CEE})$.

For a path $P \equiv \mathrm{UA} \rightarrow^+ \varsigma$, we define $\mathrm{UA} \in CE^*(\varsigma)$ if:

1. $\mathrm{UA} = CE(\varsigma)$; *or*
2. $P \equiv \mathrm{UA} \rightarrow^+ \mathrm{UEE} \rightarrow \mathrm{UA}_0 \rightarrow^* \varsigma$, $\mathrm{UA} = CE(\mathrm{UEE})$, *and* $\mathrm{UA}_0 \in CE^*(\varsigma)$.

The corresponding entry of a state ς is the procedure entry state UA that can reach ς through function calls balanced precisely by returns. A same-context entry of a state ς is a procedure entry state UA that can reach ς through tail calls as well.

This joint definition describes paths with well-behaved call–return structure. A key result of CFA2 is that, if a path is *push-monotonic*, we can *decompose* it according to this structure.

Definition 2 (Push Monotonicity). *A path $P \equiv \mathrm{UA} \rightarrow^* \varsigma$ is* push monotonic *if no state ς' such that $\mathrm{UA} \rightarrow^+ \varsigma' \rightarrow^* \varsigma$ has form $(c, \mathbf{d}, ve, t)$ where c is the continuation argument of UA.*

All paths we consider are rooted at $\mathcal{I}(pr, \mathbf{d})$ making them push-monotonic by definition. Hence, by the following theorem, they decompose into one of a handful of forms.

Theorem 1 (Path Decomposition [16]). *If $P \equiv \mathrm{UA} \rightarrow^* \varsigma$ is push monotonic, then either*

1. $\mathrm{UA} = CE(\varsigma)$ *and* $CE^*(\varsigma) = \{\mathrm{UA}\}$;
2. $P \equiv \mathrm{UA}_1 \rightarrow^+ \mathrm{UEE}_1 \rightarrow \ldots \rightarrow \mathrm{UA}_n \rightarrow^+ \mathrm{UEE}_n \rightarrow \mathrm{UA} \rightarrow^* \varsigma$ *where* $n > 0$, $\mathrm{UA}_i = CE(\mathrm{UEE}_i)$, $\mathrm{UA} = CE(\varsigma)$ *and* $CE^*(\varsigma) = \{\mathrm{UA}_1, \ldots, \mathrm{UA}_n, \mathrm{UA}\}$;
3. $P \equiv \mathrm{UA}_* \rightarrow^+ \mathrm{UEI} \rightarrow \mathrm{UA} \rightarrow^* \varsigma$ *where* $\mathrm{UA} = CE(\varsigma)$ *and* $CE^*(\varsigma) = \{\mathrm{UA}\}$;
4. $P \equiv \mathrm{UA}_0 \rightarrow^+ \mathrm{UEI} \rightarrow \mathrm{UA}_1 \rightarrow^+ \mathrm{UEE}_1 \rightarrow \ldots \rightarrow \mathrm{UA}_n \rightarrow^+ \mathrm{UEE}_n \rightarrow \mathrm{UA} \rightarrow^* \varsigma$ *where* $n > 0$, $\mathrm{UA}_i = CE(\mathrm{UEE}_i)$, $\mathrm{UA} = CE(\varsigma)$ *and* $CE^*(\varsigma) = \{\mathrm{UA}_1, \ldots, \mathrm{UA}_n, \mathrm{UA}\}$;

The upshot of path decomposition is that we can relate states that have related environments *without* appealing to those environments directly. To assist in this task, we formally define an *invocation step*.

Definition 3 (Invocation Step). *For path $P \equiv \varsigma \rightarrow^+ \varsigma'$, $\varsigma \Rightarrow \varsigma'$ if:*

1. $\varsigma \rightarrow \varsigma'$, $\varsigma \notin \mathit{UserEval}$, *and* $\varsigma \notin \mathit{ContEvalExit}$; *or*
2. $\varsigma \rightarrow \mathrm{UA} \rightarrow^+ \mathrm{CEE} \rightarrow \varsigma'$, $\varsigma \in \mathit{UserEvalInner}$, *and* $\mathrm{UA} \in CE^*(\mathrm{CEE})$.

An invocation step connects an inner state to the successive state in that invocation. This connection either corresponds directly to a step in the standard semantics (clause 1) or bridges a balanced call and return (clause 2).

Path decomposition is concerned with the call–return structure of a path. However, from our definitions, it follows that, for $\mathrm{E} \Rightarrow \mathrm{A} \Rightarrow \mathrm{E}'$, $\beta_{\mathrm{E}'}$ extends β_{E}. That is, there is an inherent connection between the call–return structure and environment structure of a path. We will use this connection in Section 7 to apply ΔCFA's environment theory to decomposed paths.

5.4 Abstract Semantics

CFA2 uses abstract interpretation to perform its flow analysis. Accordingly, it defines an *abstract semantics* which replaces the binding and value environments with a stack and heap and discards machine times completely. In this semantics, the stack is composed of frames, each of which contains both a local environment (devoid of times) and a return point. The heap stores values that may outlive the local environment of their birth.

A consequence of this abstraction is the introduction of imprecision in the semantics. Specifically, multiple abstract procedures may flow to the operator of a call, making procedure calls non-deterministic in general. Nevertheless, the abstract semantics is sound with respect to the concrete standard semantics. In this context, soundness means that, for each path taken by the concrete semantics, there is some path taken by the abstract semantics which abstracts it, state for state.

The abstract state space retains the features on which state classification depends and has corresponding domains for each domain in the concrete semantics (*UserEval*, *ContEvalExit*, etc.). In fact, a critical feature of abstract evaluation paths is that they decompose just as concrete evaluation paths do. For us, this feature means that any results we obtain solely from the decomposition of a concrete evaluation path immediately apply to abstract evaluation paths as well.

We decorate the domains, domain constituents, and metafunctions of the abstract semantics with a hat $\hat{\ }$ and denote the abstract evaluation relation with $\rightsquigarrow$.

5.5 Summarization

The final step of CFA2 is to *summarize* the abstract semantics. Summarization builds relations between program points that hold, in some sense, regardless of the stack. Accordingly, summarization operates on a semantics that itself does not model the stack. This semantics, the *local semantics*, is merely the abstract semantics with the stack and stack-dependent transitions excised. (We decorate the

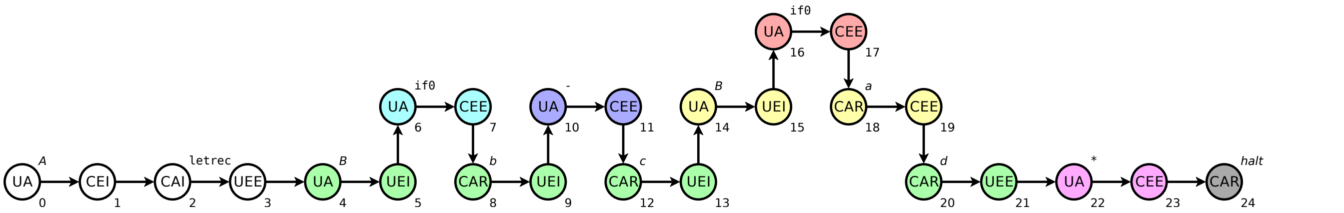

Figure 8. Decomposed evaluation path of $\mathrm{fact}(1)$

domains, domain constituents, and metafunctions of the local semantics with a tilde $\tilde{\ }$ and denote the local evaluation relation with $\widetilde{\leadsto}$.)

We call the removal of the stack from an abstract state *local abstraction* and let $|\hat{\varsigma}|_{al}$ denote the local state $\tilde{\varsigma}$ that locally abstracts $\hat{\varsigma}$. For a given program, there may be arbitrarily many reachable states that are identical modulo the stack and, hence, that $\tilde{\varsigma}$ locally abstracts.

Summarization builds five intra-path relations over local states: Seen, $\mathit{Summaries}$, $\mathit{Callers}$, $\mathit{TCallers}$, and Final.

- For each $(\widetilde{\mathrm{UA}}, \tilde{\varsigma}) \in \mathit{Seen}$, $\widetilde{\mathrm{UA}}$ is the corresponding entry of $\tilde{\varsigma}$.
- For each $(\widetilde{\mathrm{UA}}, \widetilde{\mathrm{CEE}}) \in \mathit{Summaries}$, $\widetilde{\mathrm{UA}}$ is a same-context entry of $\tilde{\varsigma}$.
- For each $(\widetilde{\mathrm{UA}}, \widetilde{\mathrm{UEI}}, \widetilde{\mathrm{UA}}') \in \mathit{Callers}$, $\widetilde{\mathrm{UA}}$ is the corresponding entry of $\widetilde{\mathrm{UEI}}$ and $\widetilde{\mathrm{UA}}'$ is the successor of $\widetilde{\mathrm{UEI}}$. Similar relations hold for each $(\widetilde{\mathrm{UA}}, \widetilde{\mathrm{UEE}}, \widetilde{\mathrm{UA}}') \in \mathit{TCallers}$.
- For each $\widetilde{\mathrm{CAR}} \in \mathit{Final}$, $\widetilde{\mathrm{CAR}}$ is a final state, i.e., applies *halt* to an argument vector.

The summarization algorithm is sound with respect to the local semantics, meaning that every abstract path has a corresponding path through the control-flow graph.

Theorem 2 (Summarization Soundness [16]). *After summarization,*

1. *if* $\hat{P} \equiv \hat{\mathcal{I}}(pr, \hat{\mathbf{d}}) \leadsto^* \widehat{\mathrm{UA}} \leadsto^* \hat{\varsigma}$ *such that* $\widehat{\mathrm{UA}} = \mathit{CE}_{\hat{P}}(\hat{\varsigma})$, *then* $(|\widehat{\mathrm{UA}}|_{al}, |\hat{\varsigma}|_{al}) \in \mathit{Seen}$*;*
2. *if* $\hat{P} \equiv \hat{\mathcal{I}}(pr, \hat{\mathbf{d}}) \leadsto^* \widehat{\mathrm{UA}} \leadsto^+ \widehat{\mathrm{CEE}}$ *such that* $\widehat{\mathrm{UA}} \in \mathit{CE}^*_{\hat{P}}(\widehat{\mathrm{CEE}})$, *then* $(|\widehat{\mathrm{UA}}|_{al}, |\widehat{\mathrm{CEE}}|_{al}) \in \mathit{Seen}$*; and*
3. *if* $\hat{P} \equiv \hat{\mathcal{I}}(pr, \hat{\mathbf{d}}) \leadsto^* \hat{\varsigma}$ *such that* $\hat{\varsigma}$ *is a final state, then* $|\hat{\varsigma}|_{al} \in \mathit{Final}$*.*

Lemma 1. *After summarization, if*

$$\hat{P} \equiv \hat{\mathcal{I}}(pr, \hat{\mathbf{d}}) \leadsto^* \widehat{\mathrm{UA}} \leadsto^+ \widehat{\mathrm{UEI}} \leadsto \widehat{\mathrm{UA}}'$$

such that $\widehat{\mathrm{UA}} = \mathit{CE}(\widehat{\mathrm{UEI}})$, *then*

$$(|\widehat{\mathrm{UA}}|_{al}, |\widehat{\mathrm{UEI}}|_{al}, |\widehat{\mathrm{UA}}'|_{al}) \in \mathit{Callers}.$$

A similar lemma holds for $\mathit{TCallers}$ soundness as well.

5.6 Control-flow Graph Reconstruction

Taken together, the five relations built by summarization combined with the local semantics define a control-flow graph of a program. Figure 9 presents the CFA2-built control-flow graph of the factorial program applied to $\top_{nat}$, the abstract value representing *any* natural number. Solid-line arrows indicate intra-procedural control flow while dashed-line arrows indicate inter-procedural control flow. This distinction is necessary as, in the abstract, inter-procedural control may flow from an invocation to itself; correspondingly, a node's vertical position no longer reliably corresponds to its denoted state's stack height. A node with multiple successors indicates non-deterministic evaluation inherent to the abstracted setting. Valid paths through this graph are those that respect a control stack: inner calls require a return point to be pushed on the stack to be consulted at continuation call.

For convenience, we derive two additional relations from the control-flow graph.

We define the first relation, $\mathit{Returns}$, as

$$\begin{aligned}\mathit{Returns} = \{(\widetilde{\mathrm{CEE}}, \widetilde{\mathrm{CAR}}) : {} & (\widetilde{\mathrm{UA}}_0, \widetilde{\mathrm{UEI}}, \widetilde{\mathrm{UA}}_1) \in \mathit{Callers},\\ & (\widetilde{\mathrm{UA}}_1, \widetilde{\mathrm{CEE}}) \in \mathit{Summaries},\\ & \widetilde{\mathrm{CAR}} = \mathit{Return}(\widetilde{\mathrm{UA}}_0, \widetilde{\mathrm{UEI}}, \widetilde{\mathrm{UA}}_1, \widetilde{\mathrm{CEE}})\}.\end{aligned}$$

This definition uses Return to synthesize a return state for each caller $(\widetilde{\mathrm{UA}}_0, \widetilde{\mathrm{UEI}}, \widetilde{\mathrm{UA}}_1)$ given summary $(\widetilde{\mathrm{UA}}_1, \widetilde{\mathrm{CEE}})$. Return synthesizes such states in precisely the same way as Update of CFA2—it simply doesn't interact with the workset as Update does. This relation makes the connection between exit states and their corresponding return states explicit.

We define the second relation, Pred, in terms of $\mathit{Returns}$ as

$$\begin{aligned}\mathit{Pred} = {} & \{(\widetilde{\mathrm{UEI}}, \widetilde{\mathrm{CAR}}) : (\widetilde{\mathrm{UA}}_0, \widetilde{\mathrm{UEI}}, \widetilde{\mathrm{UA}}_1) \in \mathit{Callers},\\ & \qquad (\widetilde{\mathrm{UA}}_1, \widetilde{\mathrm{CEE}}) \in \mathit{Summaries},\\ & \qquad (\widetilde{\mathrm{CEE}}, \widetilde{\mathrm{CAR}}) \in \mathit{Returns}\}\\ \cup \quad & \{(\tilde{\varsigma}, \tilde{\varsigma}') : (\widetilde{\mathrm{UA}}, \tilde{\varsigma}) \in \mathit{Seen},\\ & \qquad \tilde{\varsigma} \widetilde{\leadsto} \tilde{\varsigma}',\\ & \qquad \tilde{\varsigma}' \notin \widetilde{\mathit{UserApply}}\}.\end{aligned}$$

This relation allows us to both jump over a balanced call–return pair and take a single step as we traverse the control-flow graph.

6. ΔCFA

The goal of an environment analysis is to determine when two references to some variable u in fact reference the same *binding* of u. One approach to achieving this would be to determine when the two bindings were introduced: if introduced at the same point in computation (and, hence, map u to the same timestamp in their respective environments), they are the same binding. This approach is fine with concrete times that are perfectly precise, but not for abstract times which regularly conflate distinct concrete times. In consequence, an analysis that takes such an approach can answer "may be equivalent" but not "must be equivalent" binding queries.

Traditional ΔCFA comes at the problem slightly differently. Instead of tracking computation over time, it tracks the actions of a conceptual stack used to manage the environment. Then, instead of locating the introduction of bindings temporally within evaluation, it locates their introduction spatially on a representation of the stack's motion over time. Using its equipped theory to connect stack behavior to the environment, ΔCFA can turn a conservative account of stack motion into the answer to a "must be equivalent" binding query.

6.1 The Stack Behavior of a CPS Language

When a compiler of a CPS language has some means to distinguish between user- and continuation-world entities—e.g., the syntactic

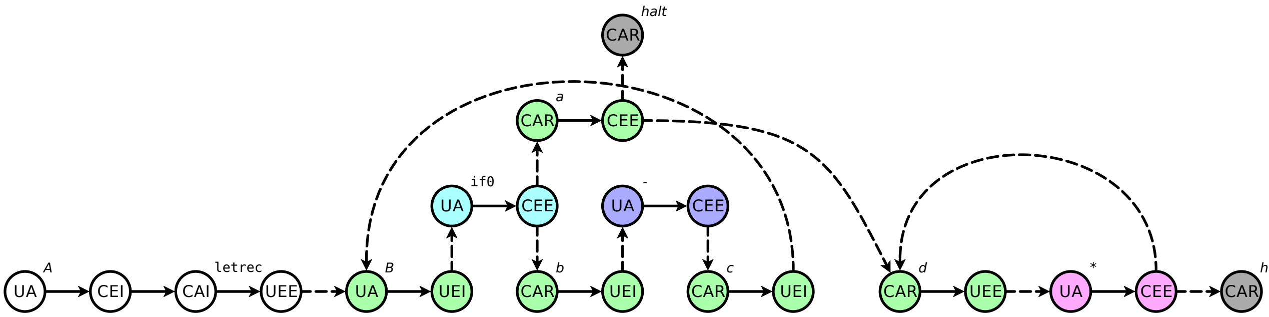

Figure 9. Control-flow graph for fact

partition of our language—it can manage the continuation using a stack rather than a heap. In such a setup, the stack houses both the code pointer and environment bindings of the continuation. When a procedure is called, a stack frame is pushed to house the bindings of its arguments. When a procedure returns, its frames are popped as its environment bindings expire and a frame is pushed on behalf of its calling procedure to house the returned result's binding.

The possibility of using a run-time stack to host the continuation closure was realized fairly early. For instance, both Rabbit [14] and Orbit [1], two early Scheme compilers that employed a CPS intermediate representation, used one. (Orbit accomplished this feat in the presence of the continuation-reifying `call/cc` by dynamically migrating the stack to the heap when it was called.)

6.2 Frame Strings

ΔCFA records stack motion using *frame strings* which have the state space F where

$$p, q \in F = \Phi^*$$
$$\phi \in \Phi ::= \langle{}^{\psi}_{t}| \mid |{}^{\psi}_{t}\rangle$$

A frame string p is a sequence of characters each of which represents an atomic stack action. A *frame character* ϕ has one of two orientations, frame push $\langle{}^{\cdot}_{\cdot}|$ or frame pop $|{}^{\cdot}_{\cdot}\rangle$, and is annotated with a label ψ and time t. Two frame characters with the same label and time but different orientations are said to be *complementary*. The empty frame string is denoted ϵ.

Three operations on frame strings will be useful: concatentation, the *net*, inversion.

1. Frame strings p and q are concatenated, denoted $p + q$, by appending their constituent frame characters. For example, $\langle{}^{\ell}_{t_1}||{}^{\ell}_{t_1}\rangle + \langle{}^{\gamma}_{t_2}| = \langle{}^{\ell}_{t_1}||{}^{\ell}_{t_1}\rangle\langle{}^{\gamma}_{t_2}|$.
2. The *net* operation $\lfloor\cdot\rfloor$ recursively annihilates adjacent complementary frame characters. For example, $\lfloor\langle{}^{\psi}_{t}||{}^{\psi}_{t}\rangle\rfloor = \epsilon$ just as
$$\lfloor\langle{}^{\psi}_{t_0}|\langle{}^{\psi'}_{t_1}||{}^{\psi'}_{t_1}\rangle|{}^{\psi}_{t_0}\rangle\rfloor = \epsilon.$$
3. The inverse ϕ^{-1} of frame character ϕ is the complementary frame character with the same label and time. For example, $\langle{}^{\psi}_{t}|^{-1} = |{}^{\psi}_{t}\rangle$. The inverse p^{-1} of frame string p is the inverse of its frame characters in reverse order. So, if $p = \phi_1 \ldots \phi_n$, then $p^{-1} = \phi_n^{-1} \ldots \phi_1^{-1}$.

Finally, we define Lab by

$$Lab : F \to \mathcal{P}(Lab)$$
$$Lab(p) = \{\psi : \langle{}^{\psi}_{t}| \in \lfloor p\rfloor\} \cup \{\psi : |{}^{\psi}_{t}\rangle \in \lfloor p\rfloor\}$$

which produces the set of labels that appear in the net of a given frame string.

6.3 Delta Frame Strings

Frame strings can represent the stack motion across a single transition, several transitions in succession, or even the entire evaluation history. When using them to represent stack motion in the first two cases, we will often use the term *delta frame string*, denoted p_Δ, to emphasize that they represent an incremental change to the entire stack history. The delta frame string across successive transitions is merely the concatenation of the delta frame strings arising from each individual transition.

Given times t_1 and t_2 in the context of program pr applied to argument vector $\mathbf{d}$, we let $[t_1, t_2]_{pr,\mathbf{d}}$ denote the delta frame string for the computation intervening those times. (We omit the subscript when pr and $\mathbf{d}$ are apparent.)

To get a better picture of which frame strings arise during evaluation, let's consider them with respect to the factorial program of Figure 5 applied to 1. Figure 8 depicts the resultant (decomposed) evaluation path in which each node is annotated by the timestamp of the represented state. The following table records the frame string behavior of this evaluation.

Net Frame String	Call Site	Δ Frame String
		$\langle{}^{A}_{1}|$
$\langle{}^{A}_{1}|$	`(letrec ...)`	$\langle{}^{\texttt{letrec}}_{3}|$
$\langle{}^{A}_{1}|\langle{}^{\texttt{letrec}}_{3}|$	`(fact m k0)`	$|{}^{\texttt{letrec}}_{3}\rangle|{}^{A}_{1}\rangle\langle{}^{B}_{5}|$
$\langle{}^{B}_{5}|$	`(if0 n ...)`	$\langle{}^{\texttt{if0}}_{7}|$
$\langle{}^{B}_{5}|\langle{}^{\texttt{if0}}_{7}|$	`if0` internal	$|{}^{\texttt{if0}}_{7}\rangle\langle{}^{b}_{9}|$
$\langle{}^{B}_{5}|\langle{}^{b}_{9}|$	`(- n 1 ...)`	$\langle{}^{\texttt{-}}_{11}|$
$\langle{}^{B}_{5}|\langle{}^{b}_{9}|\langle{}^{\texttt{-}}_{11}|$	`-` internal	$|{}^{\texttt{-}}_{11}\rangle\langle{}^{c}_{13}|$
$\langle{}^{B}_{5}|\langle{}^{b}_{9}|\langle{}^{c}_{13}|$	`(fact n-1 ...)`	$\langle{}^{B}_{15}|$
$\langle{}^{B}_{5}|\langle{}^{b}_{9}|\langle{}^{c}_{13}|\langle{}^{B}_{15}|$	`(if0 n ...)`	$\langle{}^{\texttt{if0}}_{17}|$
$\langle{}^{B}_{5}|\langle{}^{b}_{9}|\langle{}^{c}_{13}|\langle{}^{B}_{15}|\langle{}^{\texttt{if0}}_{17}|$	`if0` internal	$|{}^{\texttt{if0}}_{17}\rangle\langle{}^{a}_{19}|$
$\langle{}^{B}_{5}|\langle{}^{b}_{9}|\langle{}^{c}_{13}|\langle{}^{B}_{15}|\langle{}^{a}_{19}|$	`(k1 1)`	$|{}^{a}_{19}\rangle|{}^{B}_{15}\rangle\langle{}^{d}_{21}|$
$\langle{}^{B}_{5}|\langle{}^{b}_{9}|\langle{}^{c}_{13}|\langle{}^{d}_{21}|$	`(* n a k1)`	$|{}^{d}_{21}\rangle|{}^{c}_{13}\rangle|{}^{b}_{9}\rangle|{}^{B}_{5}\rangle\langle{}^{*}_{23}|$
$\langle{}^{*}_{23}|$	`*` internal	$|{}^{*}_{23}\rangle$

Each line of this table represents the effect of the two-step *Eval–Apply–Eval* transition, with the exception of the first and last which represent those of the single-step *Apply–Eval* and *Eval–Apply* transitions, respectively. The "Net Frame String" column contains the net of the entire frame string history. In effect, it provides a picture of the stack at each *Eval* state. The "Call Site" column contains the call site that has focus in each state. The "Δ Frame String" column contains the delta frame string that arises in the transition from that state.

Applying the program to an argument has the same frame string effect as procedure entry; the $\langle{}^{A}_{1}|$ represents the frame pushed to house the binding for `m`. We treat the `(letrec ...)` form as described in Section 4 but omit stack behavior due to the fixed-point combinator for clarity; hence, the delta frame string is $\langle{}^{\texttt{letrec}}_{3}|$. `(fact m k0)` represents a tail call in the source program—a form of procedure exit. At exit, the environment bindings of the invocation expire and their enclosing frames are popped before an environment frame for the called procedure is pushed. That these frames are contiguous and on the top of the stack is not mere happenstance; it is a consequence of the semantics of environments and the stack management policy necessary to implement it. Upon entry to the called `fact`, the call `(if0 n ...)` has focus. By expressing `if0` as a primitive procedure call with *clam*-shaped continuation argument expressions, we are construing it as a source-program inner call. This view makes sense in our context since work remains in `fact` after the work of the `if0` call (which merely discriminates `n`). As an inner call, the environment frames of its caller, `fact`, must be left intact for its return. Hence, no frames are popped before `if0`'s frame is pushed. When `if0` returns, this frame is popped and the frame $\langle{}^{b}_{9}|$ is pushed to bind `if0`'s nonexistent result value.[3]

The remaining evaluation proceeds similarly and the entire stack history of this evaluation

$$\langle{}^{A}_{1}|\langle{}^{\texttt{letrec}}_{3}\|{}^{\texttt{letrec}}_{3}\rangle|{}^{A}_{1}\rangle\langle{}^{B}_{5}|\langle{}^{\texttt{if0}}_{7}\|{}^{\texttt{if0}}_{7}\rangle\langle{}^{b}_{9}|\langle{}^{\texttt{-}}_{11}\|{}^{\texttt{-}}_{11}\rangle\langle{}^{c}_{13}|$$
$$\langle{}^{B}_{15}|\langle{}^{\texttt{if0}}_{17}\|{}^{\texttt{if0}}_{17}\rangle\langle{}^{a}_{19}\|{}^{a}_{19}\rangle|{}^{B}_{15}\rangle\langle{}^{d}_{21}\|{}^{d}_{21}\rangle|{}^{c}_{13}\rangle|{}^{b}_{9}\rangle|{}^{B}_{5}\rangle\langle{}^{\texttt{*}}_{23}\|{}^{\texttt{*}}_{23}\rangle$$

can be obtained by merely concatenating the delta frame strings of each step.

From this example, we can begin to see the connection between the shape of the delta frame string and the state classification of Section 5. *Apply* states labelled UA, CAI, or CAR always cause a single frame to be pushed. *EvalExit* states labelled UEE or CEE always cause all frames up to and including the exited procedure entry frame to be popped. Because of the balanced nature of calls and returns, only frames belonging to that procedure sit above it on the stack. *EvalInner* states labelled UEI or CEI induce no stack change since all environment frames must be preserved in case the call they perform returns.

6.4 An Environment Theory

The environment theory of ΔCFA hinges on two observations.

The first observation is that, given environment β, some later environment β', and some variable u with a binding in each environment,

$$\lfloor[\beta(u), \beta'(u)]\rfloor = \epsilon \text{ only if } \beta(u) = \beta'(u).$$

In other words, if the net delta frame string between the introduction of two bindings of the same variable is empty, they are in fact the same binding. This fact turns a question about equality of two binding times—a precarious notion in the abstract—into one about a property of the delta frame string intervening those times.

The second observation is that these times merely locate the introduction of the binding in evaluation. That is, if we consider an evaluation path to be indexed by the states' timestamps, then $\beta(u)$ identifies u's *binding state*—the state in which that binding of u was introduced—in the path. If we can recover such states without appealing to times, we can express binding equivalence conditions the same way, allowing us to apply the full environment theory of ΔCFA in a setting without times (e.g., CFA2's abstract semantics). We accomplish this recovery in Section 7.2.1.

6.5 Higher-Order Inlining

The primary goal of ΔCFA is to establish higher-order inlining safety. To this end, ΔCFA does not apply the first observation of the previous section directly. Instead, it derives weaker conditions that are easier to meet in the abstract semantics.

These weaker conditions are expressed in terms of the *collecting semantics* of a program pr applied to arguments $\mathbf{d}$, defined as

$$\mathcal{V}_{pr} : D^* \to \mathcal{P}(State)$$
$$\mathcal{V}_{pr}(\mathbf{d}) = \{\varsigma : \mathcal{I}(pr, \mathbf{d}) \to^* \varsigma\}.$$

They also refer to the *birth time* of a closure, the timestamp of the state in which the closure was created. ΔCFA's concrete semantics imprint this timestamp on the closure itself but we have no such luxury in the standard semantics. For now, we will assume we have access to an oracle

$$OracleBirth_{pr} : D^* \times Clos \to Time$$

such that $OracleBirth_{pr}(\mathbf{d}, clos)$ yields the birth time of $clos$ in the evaluation of pr applied to $\mathbf{d}$. (In Section 7.2.1, we show how to obtain this time from the evaluation path.)

We consider two of these conditions.

Theorem 3 (LOCAL-INLINABLE [8]). *It is safe to inline the term $ulam'$ in place of procedure term f' if for every state $((f\ e^*\ q)_\ell, \beta, ve, t) \in \mathcal{V}_{pr}(\mathbf{d})$ such that $f' = f$:*

1. $\mathcal{A}(f, \beta, ve) = clos = (ulam', \beta')$*;*
2. $\mathrm{free}(ulam') \subseteq \mathrm{dom}(\beta)$*;*
3. $Lab([OracleBirth_{pr}(\mathbf{d}, clos), t]) \subseteq CLab$

Intuitively, this condition justifies inlining when none of the free variables of the closure could be rebound between its point of capture and use by demanding that only continuation frames change on net. For example, it justifies the transformation of $((\texttt{identity}\,(\lambda_\ell\,(u^*)\ call))\ e^*)$ to $((\lambda_\ell\,(u^*)\ call)\ e^*)$.

Theorem 4 (EXACT-INLINABLE [8]). *It is safe to inline the term $ulam'$ in place of procedure term f' if for every state $((f\ e^*\ q)_\ell, \beta, ve, t) \in \mathcal{V}_{pr}(\mathbf{d})$ such that $f' = f$:*

1. $\mathcal{A}(f, \beta, ve) = clos = (ulam', \beta')$*; and*
2. *for each* $u \in \mathrm{free}(ulam')$*,* $\lfloor[\beta(u), \beta'(u)]\rfloor = \epsilon$.

This condition justifies inlining when the bindings of each of the free variables can be proven equivalent in the closure and calling environments. Its frame string condition is more powerful than LOCAL-INLINABLE's but also harder to meet in the abstract.

7. Pushdown ΔCFA

The crux of Pushdown ΔCFA is that, in a given path, we can recover arbitrary delta frame strings (Section 7.1) and resolve arbitrary binding and birth states (Section 7.2) from that path's CFA2 decomposition. Together, these are sufficient to apply ΔCFA's environment theory and, ultimately, test its inlining conditions.

7.1 Delta Frame String Recovery

CFA2's path decomposition allows us to recover the delta frame string of an evaluation step, given the path of which it's a part.

- For step $\mathrm{A} \to \mathrm{E}$ where $\mathrm{A} = (L(\psi), \mathbf{d}, ve, t)$, we have $[t_{\mathrm{A}}, t_{\mathrm{E}}] = \langle{}^{\psi}_{t_{\mathrm{E}}}|$. This case doesn't require any path decomposition and covers half of all evaluation steps.
- For step $\mathrm{EI} \to \mathrm{A}$, we have $[t_{\mathrm{EI}}, t_{\mathrm{A}}] = \epsilon$. Once again, no decomposition is necessary.
- For path $\mathrm{UA} \to \mathrm{EI}_1 \Rightarrow \mathrm{CA}_1 \to \mathrm{EI}_2 \Rightarrow \cdots \Rightarrow \mathrm{CA}_n \to \mathrm{EE} \to \mathrm{A}$ where

$$\begin{aligned} \mathrm{UA} &= (L(\ell), \mathbf{d}_0, ve_0, t_0) \\ \mathrm{CA}_1 &= (L(\gamma_1), \mathbf{d}_1, ve_1, t_1) \\ &\ldots \\ \mathrm{CA}_n &= (L(\gamma_n), \mathbf{d}_n, ve_n, t_n), \end{aligned}$$

[3] In an actual stack-based implementation of a CPS language, such frames are unnecessary. In our context, they are harmless and the uniformity they provide simplifies our analysis.

we have $[t_{\textsc{ee}}, t_{\textsc{a}}] = \left|{}^{\gamma_n}_{t_{\textsc{ee}}}\right\rangle \cdots \left|{}^{\gamma_1}_{t_{\textsc{ei}_2}}\right\rangle\left|{}^{\ell}_{t_{\textsc{ei}_1}}\right\rangle$. This case is an application of our observation that, at procedure exit, the frames of the exiting procedure are contiguous at the top of the stack.

The *log semantics* of ΔCFA have explicity machinery for calculating delta frame strings which provides a ground truth. In a technical report [5], we prove that the delta frame strings born of decomposition agree with those calculated by the log semantics.

7.2 Binding- and Birth-State Resolution

7.2.1 Binding-State Resolution

Let $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{e}$ and $t = \beta_{\textsc{e}}(u)$ for some variable u. Suppose we wish to determine u's binding state ς_t without consulting $\beta_{\textsc{e}}$ and let $\mathit{Bind}(P, u)$ denote the path rooted at $\mathcal{I}(pr, \mathbf{d})$ ending in this binding state. We can start to determine $\mathit{Bind}(P, u)$ by decomposing P as $\mathcal{I}(pr, \mathbf{d}) \to^{*} \textsc{a} \to \textsc{e}$ and considering the applied procedure $clos$ of $\textsc{a}$.

If $clos$ binds u, then $\textsc{e}$ is u's binding state—the first state in which this particular binding of u appeared—and $\mathit{Bind}(P, u) = P$.

If $clos$ doesn't bind u, then P takes one of two forms. If $\textsc{a} \in \mathit{ContApply}$, then $\textsc{a}$ has a predecessor $\textsc{e}'$ in its invocation and $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{e}' \Rightarrow \textsc{a} \to \textsc{e}$. Letting $P' \equiv \mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{e}'$, we have $\mathit{Bind}(P, u) = \mathit{Bind}(P', u)$. On the other hand, if $\textsc{a} \in \mathit{UserApply}$, then u must appear free in $clos$'s underlying λ-term. At this point, we can't resolve the binding state of u any further solely in terms of binding states.

We observe, however, that u's binding in $\beta_{\textsc{e}}$ is the same as its binding in the environment of $clos$. If we can determine the *birth state* of $clos$—the state at which $clos$ was constructed by $\mathcal{A}$—we can continue resolving the binding state of u.

7.2.2 Birth-State Resolution

Because u appears free in its enclosing λ-term and programs are closed, it must be that $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{ue} \to \textsc{a} \to \textsc{e}$ where $\textsc{ue} = ((f\ e^{*}\ q)_{\ell}, \beta_{\textsc{ue}}, ve, t)$ and, by the standard semantics, $clos = \mathcal{A}(f, \beta_{\textsc{ue}}, ve)$. Let $P_{\textsc{ue}} \equiv \mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{ue}$ and $\mathit{Birth}(P_{\textsc{ue}}, f)$ denote the path ending in the birth state of $clos$. From here, we'll consider the possible forms of f. If $f \in \mathit{ULam}$, then, by definition of $\mathcal{A}$, $\textsc{ue}$ is the birth state of $clos$ and $\mathit{Birth}(P_{\textsc{ue}}, f) = P_{\textsc{ue}}$. Otherwise, $f \in \mathit{UVar}$ and $clos$ was born before being bound to f. If we can resolve where f was bound—its binding state—we can continue to trace the genesis of $clos$. Section 7.2.1 describes how to resolve f's binding state.

Remarkably, the preceding reasoning suffices to obtain the binding state of an arbitrary reference and birth state of an arbitrary closure without appealing to binding environments or timestamps.

7.2.3 Binding- and Birth-State Resolution, Formally

Figure 10 presents the general binding- and birth-state resolution equations. For brevity, we will sometimes describe a path in terms of its final state and use a path in place of its final state (and vice versa).

Birth takes an $\textsc{e}$-terminated path P and an expression e in $call_{\textsc{e}}$ and yields the *birth path*—the path terminated by the birth state—of the value that flows to e. If e is some $ulam$, then the closure that "flows" to it is born in $\textsc{e}$; hence, the result is P. If e is some u, the *binding path* of u is resolved with Bind and the birth path of the value bound to u is resolved with $\mathit{BirthBP}$.

Once a binding path $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{e}' \to \textsc{a} \to \textsc{e}$ is obtained, a three-step process is used to correspond the variable u bound in $\textsc{e}$ with its argument expression. First, $\mathit{BirthBP}$ uses BP to obtain the binding position n of u in $\textsc{a}$'s procedure. Second, $\mathit{BirthIP}$ shifts focus to $\textsc{e}'$, the state in which the argument expression was evaluated. Third, $\mathit{BirthIE}$ uses IE to obtain the argument expression e for its expression position n. While verbose, decomposing the process in this way simplifies birth-time resolution in a control-flow graph.

Birth
For $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{e}$,

$$\mathit{Birth}(P, ulam) = P$$
$$\mathit{Birth}(P, u) = \mathit{BirthBP}(\mathit{Bind}(P, u), u)$$

$\mathit{BirthBP}$
For $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{*} \textsc{a} \to \textsc{e}$,

$$\mathit{BirthBP}(P, u) = \mathit{BirthIP}(\mathcal{I}(pr, \mathbf{d}) \to^{*} \textsc{a}, \mathit{BP}(\textsc{a}, u))$$

$\mathit{BirthIP}$
For $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{e} \to \textsc{a}$,

$$\mathit{BirthIP}(P, n) = \mathit{BirthIE}(\mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{e}, n)$$

$\mathit{BirthIE}$
For $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{e}$,

$$\mathit{BirthIE}(P, n) = \mathit{Birth}(P, \mathit{IE}(\textsc{e}, n))$$

Bind
For $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{*} \textsc{a} \to \textsc{e}$,

$$\mathit{Bind}(P, u) = P \text{ if } u \in B(\textsc{a})$$
$$\mathit{Bind}(P, u) = \mathit{Find}(P, u) \text{ if } u \notin B(\textsc{a})$$

Find
For $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{*} \textsc{a} \to \textsc{e}$,

1. if $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{*} \textsc{ua} \to \textsc{e}$, then $\mathit{Find}(P, u) = \mathit{Bind}(\mathit{BirthOP}(\mathcal{I}(pr, \mathbf{d}) \to^{*} \textsc{a}), u)$; and
2. if $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{e}' \Rightarrow \textsc{ca} \to \textsc{e}$, then $\mathit{Find}(P, u) = \mathit{Bind}(\mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{e}', u)$.

$\mathit{BirthOP}$
For $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{ue} \to \textsc{ua}$,

$$\mathit{BirthOP}(P) = \mathit{BirthOE}(\mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{ue})$$

$\mathit{BirthOE}$
For $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{ue}$,

$$\mathit{BirthOE}(P) = \mathit{Birth}(P, \mathit{OE}(\textsc{ue}))$$

Figure 10. General binding- and birth-state resolution

Bind takes an $\textsc{e}$-terminated path P and a variable u in $\beta_{\textsc{e}}$ and yields the binding path of u. If u is bound by the procedure of the predecessor state, Bind produces P. If not, Bind uses Find to find the previous time the environment was extended to try again.

Given an $\textsc{e}$-terminated path P, Find finds the previous time $\beta_{\textsc{e}}$ was extended. If the predecessor state $\textsc{a} \in \mathit{ContApply}$, Find uses path decomposition to walk back through the invocation to $\textsc{a}$'s predecessor. If instead $\textsc{a} \in \mathit{UserApply}$, Find uses $\mathit{BirthOP}$ to resolve the birth path of the currently-invoked procedure and determines the binding path of u from there.

$\mathit{BirthOP}$ uses the same process as $\mathit{BirthBP}$ to resolve the birth path. Essentially, the operator is treated as the zeroth argument to the call.

The correctness of these equations is expressed via two theorems.

Theorem 5 (Bind Correctness)**.** *If $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{e}$ where $\textsc{e} = (call, \beta, ve, t)$ and $\mathit{Bind}(P, u) = \mathcal{I}(pr, \mathbf{d}) \to^{+} \textsc{e}'$, then $\beta(u) = t_{\textsc{e}'}$.*

Theorem 6 (*Birth* Correctness)**.** *If* $P \equiv \mathcal{I}(pr, \mathbf{d}) \to^{+} \text{E}$ *where* $\text{E} = (call, \beta, ve, t)$*, and* $Birth(P, e) = \mathcal{I}(pr, \mathbf{d}) \to^{+} \text{E}'$*, then* $OracleBirth_{pr}(\mathbf{d}, \mathcal{A}(e, \beta, ve)) = t_{\text{E}'}$*.*

The standard semantics records variable binding times and the log semantics of ΔCFA records closure birth times, together providing a ground truth. In a technical report [5], we prove these equations correct with respect to this ground truth by induction over the definitions and path decomposition.

Now that we have achieved a perfect time–state correspondence, we will replace times as used by ΔCFA with the states they stamp. Thus, intervals will be $[\varsigma_0, \varsigma_1]$ for some ς_0 and ς_1.

7.3 State Resolution in a Control-Flow Graph

The binding- and birth-state resolution method of Section 7.2 assumes a linear evaluation path in which every state has exactly one predecessor (except the program entry state). A control-flow graph breaks this assumption: non-determinism gives rise to forks and joins, and recursion gives rise to cycles. However, by Theorem 2, every abstract evaluation path is a stack-respecting path through the graph. Thus, if we can show that every stack-respecting path through the graph meets a given environment condition, then every abstract evaluation path—and concrete evaluation path, by abstraction soundness—does also. This observation will govern our approach.

A given state $\tilde{\varsigma}$ in a control-flow graph typically represents a state in many abstract paths and even many states in a single abstract path. Consequently, the task is not to resolve a particular state in a single path but every state in every path $\tilde{\varsigma}$ represents. Thus, we will let $\tilde{\varsigma}$ represent the *set* of stack-respecting paths rooted at $\tilde{\mathcal{I}}(pr, \hat{\mathbf{d}})$ that reach $\tilde{\varsigma}$.

Because these paths may follow a circuit in the graph arbitrarily many times, we will instead find the fixed point of state resolution over the graph. This resolution process is symmetric to that of CFA2 summarization:

- Summarization traverses the local relation forward; resolution traverses it backward.
- Summarization keeps track of callers to connect discovered procedure exits with returns; resolution keeps track of returns to connect resolved invocations with callers.
- Summarization memoizes the control flow from an entry state to a reachable exit state; resolution memoizes net resolution from an exit state to a same-level entry. (For simplicity, the resolution function we present omits memoization.)

To ensure termination, we will not allow intraprocedural path segments to be visited multiple times in different stages of resolution. This policy admits paths in which, for example, recursive calls merely intervene the birth or binding state and the end of the path, a common feature when inlining is safe.

In a control-flow graph, resolution is incremental so we use *resolution terms* to capture the state of resolution along a path. A resolution term $\kappa[\tilde{\varsigma}] \in Z$ is a *resolution context* κ *composed* with path $\tilde{\varsigma}$. A resolution context κ, defined as

$$\begin{aligned}\kappa := {} & Birth(\kappa, e) \mid BirthBP(\kappa, u) \mid BirthIP(\kappa, n) \\ & BirthIE(\kappa, n) \mid Bind(\kappa, u) \mid Find(\kappa, u) \\ & BirthOP(\kappa) \mid BirthOE(\kappa) \mid \bullet\end{aligned}$$

denotes a stage of resolution with respect to its composed term. Composition of a resolution context with path $\tilde{\varsigma}$ replaces the "hole" $\bullet$ with $\tilde{\varsigma}$, and can be performed on other resolution contexts and terms as well. The only resolution contexts we use arise directly from the resolution equations in Figure 10.

$$H : \mathcal{S} \times \mathcal{R} \times \mathcal{F} \to \mathcal{S} \times \mathcal{R} \times \mathcal{F}$$

$$H(S, R, F) = \bigoplus_{(\kappa[\tilde{\varsigma}], \kappa_0[\tilde{\text{E}}]) \in S} Inner(S, R, F, \kappa[\tilde{\varsigma}], \kappa_0[\tilde{\text{E}}])$$

$$Inner(S, R, F, \kappa[\tilde{\varsigma}], \kappa_0[\tilde{\text{E}}]) =$$

$$\begin{cases} Link(S, R, F, \kappa[\tilde{\varsigma}], \kappa_0[\tilde{\text{E}}]) & \text{if } \kappa = \kappa'[\gamma] \\ (S, R, F \oplus \{\tilde{\varsigma}\}) & \text{if } \kappa = \bullet \\ Inner(S, R, F, \text{resolve}(\kappa[\tilde{\varsigma}]), \kappa_0[\tilde{\text{E}}]) & \text{otherwise} \end{cases}$$

$$Link(S, R, F, \kappa[\tilde{\text{A}}], \kappa_0[\tilde{\text{E}}]) =$$

$$\begin{cases} LinkCall(S, R, F, \kappa[\tilde{\text{A}}], \kappa_0[\tilde{\text{E}}]) & \text{if } \tilde{\text{A}} \in \widetilde{UserApply} \\ CheckRetr(S, R, F, \kappa[\tilde{\text{A}}], \kappa_0[\tilde{\text{E}}]) & \text{if } \tilde{\text{A}} \in \widetilde{ContApply} \end{cases}$$

$$LinkCall(S, R, F, \kappa[\widetilde{\text{UA}}], \kappa_0[\tilde{\text{E}}]) =$$

$$\begin{cases} (S \oplus LocalCall(R, \kappa[\widetilde{\text{UA}}], \kappa_0[\tilde{\text{E}}]), R, F) & \tilde{\text{E}} \in \widetilde{CEvalExit} \\ (S \oplus GlobalCall(\kappa[\widetilde{\text{UA}}], \kappa_0[\tilde{\text{E}}]), R, F) & \tilde{\text{E}} \in \widetilde{UserEval} \end{cases}$$

$$LocalCall(R, \kappa[\gamma[\widetilde{\text{UA}}]], \kappa_0[\widetilde{\text{CEE}}]) =$$

$$\begin{aligned} \{(\kappa[link(\gamma)[\widetilde{\text{UEI}}]], \kappa_2[\tilde{\text{E}}]) : {} & (\kappa_0[\widetilde{\text{CEE}}], \kappa_1[\widetilde{\text{CAR}}], \kappa_2[\tilde{\text{E}}]) \in R, \\ & (\widetilde{\text{UEI}}, \widetilde{\text{CAR}}) \in Pred, \\ & (\widetilde{\text{UA}}_0, \widetilde{\text{UEI}}, \widetilde{\text{UA}}) \in Callers\} \end{aligned}$$

$$\cup$$

$$\{(\kappa[link(\gamma)[\widetilde{\text{UEE}})], \kappa_0[\widetilde{\text{CEE}}]) : (\widetilde{\text{UA}}_0, \widetilde{\text{UEE}}, \widetilde{\text{UA}}) \in TCallers\}$$

$$GlobalCall(\kappa[\gamma[\widetilde{\text{UA}}]], \kappa_0[\widetilde{\text{UE}}_0]) =$$

$$\begin{aligned} \{(\kappa[link(\gamma)[\widetilde{\text{UE}}]], \kappa[link(\gamma)[\widetilde{\text{UE}}]]) : & \\ & (\widetilde{\text{UA}}', \widetilde{\text{UE}}, \widetilde{\text{UA}}) \in Callers \cup TCallers\} \end{aligned}$$

$$CheckRetr(S, R, F, \kappa[\gamma[\widetilde{\text{CA}}]], \kappa_0[\tilde{\text{E}}]) =$$

$$\begin{cases} Inner(S, R, F, \kappa[link(\gamma)[\widetilde{\text{CEI}}]], \kappa_0[\tilde{\text{E}}]) & \text{if } (\widetilde{\text{CEI}}, \widetilde{\text{CA}}) \in Pred \\ (S, R \oplus LinkRetrs(\kappa[\gamma[\widetilde{\text{CA}}]], \kappa_0[\tilde{\text{E}}]), F) & \text{if } (\widetilde{\text{UEI}}, \widetilde{\text{CA}}) \in Pred \end{cases}$$

$$LinkRetrs(\kappa[\gamma[\widetilde{\text{CAR}}]], \kappa_0[\tilde{\text{E}}]) =$$

$$\{(\kappa[link(\gamma)[\widetilde{\text{CEE}}]], \kappa[\gamma[\widetilde{\text{CAR}}]], \kappa_0[\tilde{\text{E}}]) : (\widetilde{\text{CEE}}, \widetilde{\text{CAR}}) \in Returns\}$$

Figure 11. Resolution function H

To record the resolution on each path segment, we pair each resolution term $\kappa[\tilde{\varsigma}]$ with an "anchor" resolution term $\kappa_0[\tilde{\text{E}}]$ which, with $\kappa[\tilde{\varsigma}]$, defines the span of the resolved segment. We term these pairs *resolution segments*. In addition to facilitating the link between the resolution of interprocedural path segments, the anchor state provides a small degree of polyvariance to the resolution process.

Resolution is expressed as a fixed point of the function H, defined in Figure 11. H operates over triples $(S, R, F) \in \mathcal{S} \times \mathcal{R} \times \mathcal{F}$ where

$$\mathcal{S} = \mathcal{P}(Z \times Z) \qquad \mathcal{R} = \mathcal{P}(Z \times Z \times Z) \qquad \mathcal{F} = \mathcal{P}(\widetilde{State}).$$

Given a resolution to calculate of the form $\kappa[\widetilde{\text{UE}}]$, we seed the fixed-point finder of H with $(S_0, \emptyset, \emptyset)$ where $S_0 = \{(\kappa[\widetilde{\text{UE}}], \kappa[\widetilde{\text{UE}}])\}$.

H is defined as the *fixed-point join* (via the $\oplus$ operator) of *Inner* mapped over the accumulating set S. For relations X_0 and X_1, $X_0 \oplus X_1$ is merely the union of relations X_0 and X_1 with element equality modulo resolution contexts. If the resulting relation would relate states $\tilde{\varsigma}$ and $\tilde{\varsigma}'$ in multiple distinct ways, the union "fails" and communicates this failure to the fixed-point finder. For example, $\{(\kappa[\tilde{\varsigma}], \kappa_0[\tilde{\text{E}}])\} \oplus \{(\kappa'[\tilde{\varsigma}], \kappa_0[\tilde{\text{E}}])\}$ fails for $\kappa \neq \kappa'$. Fixed-point join behaves componentwise on tuples so that

$$(S_0, R_0, F_0) \oplus (S_1, R_1, F_1) = (S_0 \oplus S_1, R_0 \oplus R_1, F_0 \oplus F_1),$$

with fixed-point join over $\mathcal{F}$ devolving to set union.

$Inner$ is the workhorse of the resolution process. If it detects that its resolution term argument has reached a procedure boundary, it dispatches $Link$. If it detects that its argument is resolved, it adds it to the set of resolved states F. Otherwise, it performs one step of resolution via $resolve$. The $resolve$ function simply applies an identity from the resolution equations of Figure 10.

$Link$ connects its $\widetilde{Apply}$-focused argument to its calling states via $LinkCall$ or exiting states via $CheckRetr$, depending on whether its argument focuses on an entry or return state. If the context state of $LinkCall$'s argument is $\widetilde{\text{CEE}}$, then a resolution segment recorded in R awaits resolution of this segment and is continued with $LocalCall$. If no segment awaits, then all path prefixes must be considered with $GlobalCall$.

$CheckRetr$ determines whether a $\widetilde{ContApply}$ state is due to an inner user or continuation call. If due to a user call, $CheckRetr$ links it with its exit states via $LinkRetrs$. If due to a continuation call, $CheckRetr$ immediately links it to its local predecessor and continues resolution via $Inner$.

$LinkRetrs$ uses $Returns$ to link to the exits that reach a return state.

Several functions make use of *link contexts* γ defined

$$\gamma := BirthOP(\bullet) \;\mid\; BirthIP(\bullet, n).$$

Link contexts are a component of all resolution contexts sitting at an interprocedural boundary. The single-step resolution function $link : \gamma \to \kappa$ defined

$$\begin{aligned} link(BirthOP(\bullet)) &= BirthOE(\bullet) \\ link(BirthIP(\bullet, n)) &= BirthIE(\bullet, n). \end{aligned}$$

makes the interprocedural jump.

7.4 Applying the Inlining Conditions

Being able to resolve arbitrary binding- and birth-states in a control-flow graph, we can now look to applying the inlining conditions of ΔCFA's environment theory. We first present the abstract conditions for inlining in Pushdown ΔCFA. The soundness of the first condition rests on that of CFA2. The frame string conditions require that we express the binding and birth states in terms of our resolution vocabulary.

Theorem 7 (LOCAL-INLINABLE (Abstract))**.** *It is safe to inline the term $ulam'$ in place of procedure term f' if, for every $(\widetilde{\text{UA}}, \widetilde{\text{UE}}) \in Seen$ such that $\widetilde{\text{UE}} = ((f\ e^*\ q)_\ell, env, h)$,*

1. $\widetilde{\mathcal{A}}(f, env, h) = \{ulam'\}$,
2. $\text{free}(ulam') \subseteq Scope(\ell)$, *and*
3. $Lab([Birth(\widetilde{\text{UE}}, f), \widetilde{\text{UE}}]) \subseteq CLab$.

Theorem 8 (EXACT-INLINABLE (Abstract))**.** *It is safe to inline the term $ulam'$ in place of procedure term f' if, for every $(\widetilde{\text{UA}}, \widetilde{\text{UE}}) \in Seen$ such that $\widetilde{\text{UE}} = ((f\ e^*\ q)_\ell, env, h)$,*

1. $\widetilde{\mathcal{A}}(f, env, h) = \{ulam'\}$ *and,*
2. $\text{free}(ulam') \subseteq Scope(\ell)$, *and*
3. *for each $u \in \text{free}(ulam')$,*

$$\lfloor [Bind(\widetilde{\text{UE}}, u), Bind(Birth(\widetilde{\text{UE}}, f), u)] \rfloor = \epsilon.$$

We add the second condition of LOCAL-INLINABLE to EXACT-INLINABLE for early detection of unsafe inlinings. Without it, resolution is still sound but could walk back to program entry before it determined that some u was not in $Scope(\ell)$.

7.5 Joint Resolution

The EXACT-INLINABLE frame string condition concerns the delta frame string of two resolution terms. In order to for this condition to hold, these terms must in fact denote the same state. As our resolution framework handles resolution terms over *sets* of paths, our current strategy cannot demonstrate this. For example, both binding states may resolve to the same local state (i.e., set of paths) in a recursive invocation, but refer to two different invocations in an abstract path.

To overcome this limitation, we need to ensure that both resolution terms resolve to the same local state *at the same rate*. We achieve this by modifying H to operate over *pairs* of resolution segments

$$((\kappa[\tilde{\varsigma}], \kappa_0[\tilde{\text{E}}]), (\kappa'[\tilde{\varsigma}'], \kappa_0'[\tilde{\text{E}}']))$$

and resolving them in lock-step.

We likewise modify the domains $\mathcal{S}$ and $\mathcal{R}$, the fixed-point join operator $\oplus$, and the auxiliary functions to handle pairs. In particular, $Inner$ is modified to $resolve$ both terms together by having the separate resolutions rendezvous just before stepping to another state on the path. $Inner$ fails if only one term resolves at a particular state, which translates to a failure to meet the frame string condition.

Resolution of terms A and B may reach some $\widetilde{\text{CAR}}$ such that term A can skip over the call to the preceding $\widetilde{\text{UEI}}$ but term B must traverse it. In this case, we pause joint resolution to find a single, nested fixed point of resolution for all calls from $\widetilde{\text{UEI}}$ that return to $\widetilde{\text{CAR}}$ by seeding S with

$$\{(\kappa[\widetilde{\text{CEE}}], \kappa[\widetilde{\text{CEE}}]) : \widetilde{\text{CEE}} \in Exits(\widetilde{\text{CAR}})\}$$

for the appropriate context κ where

$$Exits(\widetilde{\text{CAR}}) = \{\widetilde{\text{CEE}} : (\widetilde{\text{CEE}}, \widetilde{\text{CAR}}) \in Returns\}.$$

This seed ensures that $GlobalCall$ will never be invoked which has the effect of cutting off resolution once all same-context entries of each $Exits(\widetilde{\text{CAR}})$ have been reached. After the fixed-point search converges, if either

- F is non-empty, meaning some term resolved within the call, or
- all $\widetilde{\text{UA}}$ such that $\widetilde{\text{UEI}} \tilde{\rightsquigarrow} \widetilde{\text{UA}}$ do not have the same resolution context κ' on all paths,

then joint resolution fails. Otherwise, joint resolution continues with $link(\kappa')$ as the context of term B.

8. Example

We will walk through our approach using the introductory example program seen in Figure 1. Figure 12 presents its control-flow graph when applied to $\top_{list}$, an abstract value representing any list.

We are interested in inlining `(λ (u) ...)` at `(f w)`. The only state at which `(f w)` has control is UEI_2 so the conditions must be met only at this state. The first two conditions of each inlining condition are easily checked, and both are met. The LOCAL-INLINABLE frame string condition is met if we can demonstrate that $Birth(\text{UEI}_2, \texttt{f}) \subseteq CLab$ which holds if $Birth(\text{UEI}_2, \texttt{f})$ resolves before UA is reached. The EXACT-INLINABLE frame string condition is met if we can demonstrate that

$$\lfloor [Bind(Birth(\text{UEI}_2, \texttt{f}), \texttt{y}), Bind(\text{UEI}_2, \texttt{y})] \rfloor = \epsilon$$

which holds if the resolution terms in each pair resolve simultaneously on all path segments.

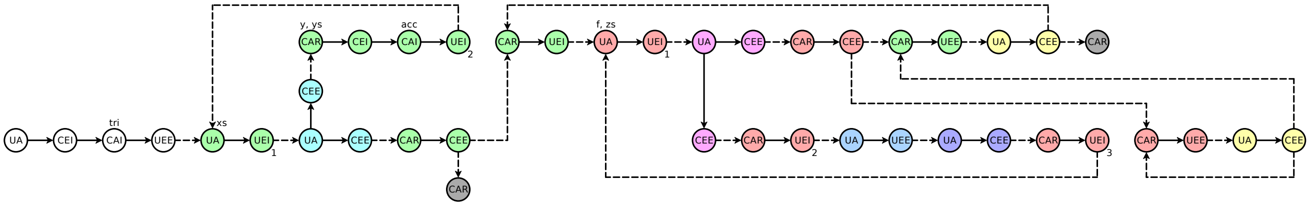

Figure 12. Control-flow graph for tri

Resolution begins with the resolution pair of the frame string condition. (For space, we abbreviate the binding- and birth-state resolution names.)

$$\begin{array}{ll}
\mathit{Bind}(\mathit{Birth}(\text{UEI}_2, \texttt{f}), \texttt{y}) & \\
\mathit{Bind}(\mathit{BBP}(\mathit{Bind}(\text{UEI}_2, \texttt{f}), \texttt{f}), \texttt{y}) & \mathit{Bind}(\text{UEI}_2, \texttt{y}) \\
\mathit{Bind}(\mathit{BBP}(\mathit{Find}(\text{UEI}_2, \texttt{f}), \texttt{f}), \texttt{y}) & \mathit{Find}(\text{UEI}_2, \texttt{y}) \\
\mathit{Bind}(\mathit{BBP}(\mathit{Bind}(\text{UEI}_1, \texttt{f}), \texttt{f}), \texttt{y}) & \mathit{Bind}(\text{UEI}_1, \texttt{y}) \\
\mathit{Bind}(\mathit{BBP}(\text{UEI}_1, \texttt{f}), \texttt{y}) & \mathit{Find}(\text{UEI}_1, \texttt{y}) \\
\mathit{Bind}(\mathit{BIP}(\text{UA}, 1), \texttt{y}) & \mathit{Bind}(\mathit{BOP}(\text{UA}), \texttt{y})
\end{array}$$

At this point, we can inspect the resolution trace to determine whether $\mathit{Birth}(\text{UEI}_2, \texttt{f})$ resolved fully. If so, the LOCAL-INLINABLE frame string condition is met and we can cut off the remainder of the process. In this case, it didn't resolve, so we must continue resolution of other path segments.

Paths arrive at UA via both an outside call and a recursive call. Resolution for path segments that arrive via the recursive call proceeds as

$$\begin{array}{ll}
\mathit{Bind}(\mathit{BIE}(\text{UEI}_3, 1), \texttt{y}) & \mathit{Bind}(\mathit{BOE}(\text{UEI}_3), \texttt{y}) \\
\mathit{Bind}(\mathit{Birth}(\text{UEI}_3, \texttt{f}), \texttt{y}) & \mathit{Bind}(\mathit{Birth}(\text{UEI}_3, \texttt{acc}), \texttt{y}) \\
\mathit{Bind}(\mathit{BBP}(\mathit{Bind}(\text{UEI}_3, \texttt{f}), \texttt{f}), \texttt{y}) & \mathit{Bind}(\text{UEI}_3, \texttt{y}) \\
\mathit{Bind}(\mathit{BBP}(\mathit{Find}(\text{UEI}_3, \texttt{f}), \texttt{f}), \texttt{y}) & \mathit{Find}(\text{UEI}_3, \texttt{y}) \\
\mathit{Bind}(\mathit{BBP}(\mathit{Bind}(\text{UEI}_2, \texttt{f}), \texttt{f}), \texttt{y}) & \mathit{Bind}(\text{UEI}_2, \texttt{y}) \\
\mathit{Bind}(\mathit{BBP}(\text{UEI}_1, \texttt{f}), \texttt{y}) & \mathit{Find}(\text{UEI}_1, \texttt{y}) \\
\mathit{Bind}(\mathit{BIP}(\text{UA}, 1), \texttt{y}) & \mathit{Bind}(\mathit{BOP}(\text{UA}), \texttt{y})
\end{array}$$

Because recursion is achieved through unfolding, $\mathit{Birth}(\text{UEI}_3, \texttt{acc})$ resolves immediately. (We have removed all trace of the combinator, however.) Having seen this resolution state pair before, no new path segments are discovered to resolve.

Resolution for path segments that arrive via UEI_2 proceeds as

$$\begin{array}{ll}
\mathit{Bind}(\mathit{BIE}(\text{UEI}_2, 1), \texttt{y}) & \mathit{Bind}(\mathit{BOE}(\text{UEI}_2), \texttt{y}) \\
\mathit{Bind}(\mathit{Birth}(\text{UEI}_2, \lambda), \texttt{y}) & \mathit{Bind}(\mathit{Birth}(\text{UEI}_2, \texttt{acc}), \texttt{y}) \\
\mathit{Bind}(\text{UEI}_2, \texttt{y}) & \mathit{Bind}(\mathit{BBP}(\mathit{Bind}(\text{UEI}_2, \texttt{acc}), \texttt{acc}), \texttt{y}) \\
\mathit{Find}(\text{UEI}_2, \texttt{y}) & \mathit{Bind}(\mathit{BBP}(\mathit{Find}(\text{UEI}_2, \texttt{acc}), \texttt{acc}), \texttt{y}) \\
\mathit{Bind}(\text{UEI}_1, \texttt{y}) & \mathit{Bind}(\mathit{BBP}(\mathit{Bind}(\text{UEI}_1, \texttt{acc}), \texttt{acc}), \texttt{y}) \\
\mathit{Find}(\text{UEI}_1, \texttt{y}) & \mathit{Bind}(\mathit{BBP}(\text{UEI}_1, \texttt{acc}), \texttt{y}) \\
 & \mathit{Bind}(\mathit{BIP}(\text{CAI}, 1), \texttt{y}) \\
 & \mathit{Bind}(\mathit{BIE}(\text{CEI}, 1), \texttt{y}) \\
 & \mathit{Bind}(\mathit{Birth}(\text{CEI}, \texttt{(}\lambda\ \texttt{(f zs) ...)}), \texttt{y}) \\
\mathit{Bind}(\text{CEI}, \texttt{y}) & \mathit{Bind}(\text{CEI}, \texttt{y}) \\
\text{CEI} & \text{CEI}
\end{array}$$

with resolution on this path segment converging. As the resolved state is the same, we have

$$\lfloor [\text{CEI}, \text{CEI}] \rfloor = \lfloor \epsilon \rfloor = \epsilon.$$

Since all path segments have been considered, the EXACT-INLINABLE frame string condition is met.

9. Related and Future Work

Vardoulakis suggested that a pushdown abstraction could increase the precision of ΔCFA [15, p. 110] as this work demonstrates. We've presented Pushdown ΔCFA in terms of CFA2 [16], but it can, in principle, be applied to other pushdown-abstraction analyses, such as PDCFA [4], AAC [7], and P4F [6]. Since we've formulated it as an a posteriori analysis, neither the implementation nor the correctness claim of any of these analyses would need to be changed to do so.

Pushdown ΔCFA leverages the environment theory of ΔCFA [8, 10] but avoids its explicit stack machinery.

Both our framework and reflow analysis [13, Ch. 8] are able to justify the inlining in the introductory example using, it seems, fundamentally different approaches. While our framework has the benefit of formal correctness, it would be useful to better understand their relative strengths and weaknesses.

Unchanged Variable Analysis (UVA) [2] operates over the control-flow graph produced by a k-CFA analysis [12, 13]. It proves bindings equivalent when there is no path in the control-flow graph between binding and use site via which it can be rebound. As an a posteriori analysis, our approach is similar in spirit. UVA suffers from the weaknesses inherent in a finite-state analysis and the authors observe that it is not as effective as k-CFA-supported ΔCFA.

Abstract counting [8, 11] can equivocate abstract bindings over the same variable when only one concrete counterpart can exist and, unlike ours, its technique is impervious to control flow. Combining the two approaches and discharging the justification to the one more equipped to handle it seems promising.

Our binding- and birth-time resolution mechanism is quite flexible, and can support analyses for both higher-order copy propagation [13] and escape analysis [8, Ch. 10].

A natural extension to this work would be to remove the restriction on the appearance of continuation references k, allowing user-world programs with `call/cc`. Vardoulakis and Shivers [17] extend CFA2 to support `call/cc` by enlarging the *corresponding entry* definition to match continuation calls with all possible binding states. This extension introduces a new kind of imprecision into the summarization algorithm: while traditional CFA2 is *complete* with respect to the abstract semantics—meaning that every stack-respecting path through the control-flow graph corresponds state-for-state to some abstract path—`call/cc`-supporting CFA2 is not: some paths through the control-flow graph have no corresponding abstract path. This imprecision certainly impedes Pushdown ΔCFA but does not render it unsound. In fact, resolution of binding and birth states of variables not bound to the result of the `call/cc` call is completely unaffected by `call/cc` calls—even intervening ones.

Acknowledgements

We thank the anonymous reviewers for their thoughtful, thorough, and constructive comments which have greatly improved this paper.

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] Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. *Orbit: An optimizing compiler for Scheme*, volume 21. ACM, 1986.

[2] Lars Bergstrom, Matthew Fluet, Matthew Le, John Reppy, and Nora Sandler. Practical and effective higher-order optimizations. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 81–93. ACM, 2014.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[4] Christopher Earl, Matthew Might, and David Van Horn. Pushdown control-flow analysis of higher-order programs. In *2010 Workshop on Scheme and Functional Programming (Scheme 2010)*, Montreal, Quebec, Canada, August 2010.

[5] Kimball Germane and Matthew Might. A posteriori environment analysis via pushdown ΔCFA. Technical report, November 2016. `http://kimball.germane.net/germane2017pddeltacfa-techreport.pdf`.

[6] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *43rd Annual ACM Symposium on Principles of Programming Languages*, POPL '16, pages 691–704, New York, NY, USA, 2016. ACM.

[7] James Ian Johnson and David Van Horn. Abstracting abstract control. In *10th ACM Symposium on Dynamic Languages*, pages 11–22. ACM, 2014.

[8] Matthew Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, June 2007.

[9] Matthew Might. Shape analysis in the absence of pointers and structure. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 263–278. Springer, 2010.

[10] Matthew Might and Olin Shivers. Environment analysis via ΔCFA. In *33rd ACM Symposium on Principles of Programming Languages*, POPL '06, pages 127–140, New York, NY, USA, 2006. ACM.

[11] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 13–25, New York, NY, USA, 2006. ACM.

[12] Olin Shivers. Control flow analysis in scheme. In *ACM 1998 Conference on Programming Language Design and Implementation*, PLDI '88, pages 164–174. ACM, 1988.

[13] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

[14] Guy L Steele Jr. Rabbit: A compiler for scheme. Technical report, Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1978.

[15] Dimitrios Vardoulakis. *Cfa2: pushdown flow analysis for higher-order languages*. PhD thesis, Northeastern University Boston, 2012.

[16] Dimitrios Vardoulakis and Olin Shivers. CFA2: a context-free approach to control-flow analysis. In *European Symposium on Programming*, pages 570–589, 2010.

[17] Dimitrios Vardoulakis and Olin Shivers. Pushdown flow analysis of first-class control. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 69–80. ACM, 2011.