Pushdown Flow Analysis of First-Class Control

Dimitrios Vardoulakis Olin Shivers

Northeastern University

dimvar@ccs.neu.edu shivers@ccs.neu.edu

Abstract

Pushdown models are better than control-flow graphs for higher-order flow analysis. They faithfully model the call/return structure of a program, which results in fewer spurious flows and increased precision. However, pushdown models require that calls and returns in the analyzed program nest properly. As a result, they cannot be used to analyze language constructs that break call/return nesting such as generators, coroutines, `call/cc`, *etc*.

In this paper, we extend the CFA2 flow analysis to create the first pushdown flow analysis for languages with first-class control. We modify the abstract semantics of CFA2 to allow continuations to escape to, and be restored from, the heap. We then present a summarization algorithm that handles escaping continuations via a new kind of summary edges. We prove that the algorithm is sound with respect to the abstract semantics.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis

General Terms Languages

Keywords pushdown flow analysis, first-class continuations, restricted continuation-passing style, summarization

1. Introduction

Function call and return is the fundamental control-flow mechanism in higher-order languages. Therefore, if a flow analysis is to model program behavior faithfully, it must handle call and return well. Pushdown models of programs [14, 16, 21] enable flow analyses with unbounded call/return matching. These analyses are more precise than analyses based on control-flow graphs.

Pushdown models require that calls and returns in the analyzed program nest properly. However, many control constructs, some of them in mainstream programming languages, break call/return nesting. *Generators* [3, Python] [2, JavaScript] are functions that are usually called inside loops to produce a sequence of values one at a time. A generator executes until it reaches a `yield` statement, at which point it returns the value passed to `yield` to its calling context. When the generator is called again, execution resumes at the first instruction after the `yield`. *Coroutines* [6, Simula67][4, Lua] can also suspend and resume their execution, but are more expressive than generators because they can specify where to pass control to when they yield. Last but not least, *first-class continuations* reify the rest of the computation as a function. Continuations allow complex control flow, such as jumping back to functions that have already returned. Continuations come in two flavors. Undelimited continuations (`call/cc` in Scheme [19] and SML/NJ [5]) capture the entire stack. Delimited continuations [7, 9] [15, Scala 2.8] capture part of the stack. Continuations can express generators and coroutines, and also multi-threading [17, 24] and Prolog-style backtracking. All these operators provide a rich variety of control behaviors. Unfortunately, we cannot currently use pushdown models to analyze programs that use them.

We rectify this situation by extending the CFA2 flow analysis [21] to languages with first-class control. We make the following contributions.

- CFA2 is based on abstract interpretation of programs in continuation-passing style (*abbrev.* CPS). We present a CFA2-style abstract semantics for Restricted CPS, a variant of CPS that allows continuations to escape but also permits effective reasoning about the stack [23]. When we detect a continuation that may escape, we copy the stack into the heap (sec. 4.3). We prove that the abstract semantics is a safe approximation of the actual runtime behavior of the program (sec. 4.4).
- In pushdown flow analysis, each state has a stack of unbounded size. Hence, the state space is infinite. Algorithms that explore the state space use a technique called *summarization*. First-class control causes the stack to be copied into the heap, so our analysis must also deal with infinitely many heaps. We show that it is not necessary to keep continuations in the heap during summarization; we handle escaping continuations using a new kind of summary edges (sec. 5.3).
- When calls and returns nest properly, execution paths satisfy a property called *unique decomposition:* for each state s in a path, we can uniquely identify another state s' as the entry of the procedure that contains s [16]. In the presence of first-class control, a state can belong to more than one procedure. We allow paths that are decomposable in multiple ways and prove that our analysis is sound (sec. 5.4).
- If continuations escape upward, a flow analysis cannot generally avoid spurious control flows. What about continuations that are only used downward, such as exception handlers or continuations captured by `call/cc` that never escape? We show that CFA2 can avoid spurious control flows for downward continuations (sec. 5.5).

2. Why pushdown models?

Finite-state flow analyses, such as k-CFA, approximate programs as graphs of abstract machine states. Each node in such a graph represents a program point plus some amount of abstracted environment and control context. Every path in the graph is considered a possible execution of the program. Thus, executions are strings in a *regular* language.

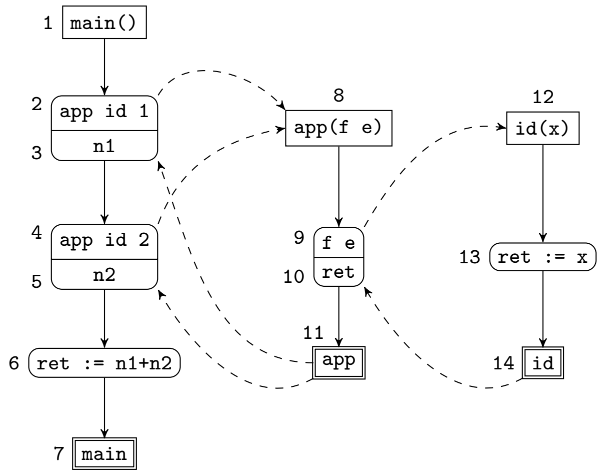

Figure 1: Control-flow graph for a simple program

Finite-state analyses do not handle call and return well. They remember only a bounded number of pending calls, so they allow paths in which a function is called from one program point and returns to a different one.

Execution traces that match calls with returns are strings in a *context-free* language. Therefore, by abstracting a program to a pushdown automaton (or equivalent), we can use the stack to eliminate call/return mismatch. The following examples illustrate the advantages of pushdown models.

2.1 Data-flow information

The following Scheme program defines the apply and identity functions, then binds `n1` to 1 and `n2` to 2 and adds them. At program point `(+ n1 n2)`, both variables are bound to constants; we would like a static analysis to be able to find that.

```
(define app
  (λ (f e) (f e)))

(define id
  (λ (x) x))

(let* ((n1 (app id 1))
       (n2 (app id 2)))
  (+ n1 n2))
```

Fig. 1 shows the control-flow graph for this program. In the graph, the top level of the program is presented as a function called `main`. Function entry and exit nodes are rectangles with sharp corners. Inner nodes are rectangles with rounded corners. Each call site is represented by a call node and a corresponding return node, which contains the variable to which the result of the call is assigned. Each function uses a local variable `ret` for its return value. Solid arrows are intraprocedural steps. Dashed arrows go from call sites to function entries and from function exits to return points. There is no edge between call and return nodes; a call reaches its corresponding return only if the callee terminates.

A monovariant analysis, such as 0CFA, considers all paths to be valid executions. Thus, we can bind `n1` to 2 by calling `app` from 4 and returning to 3. Also, we can bind `n2` to 1 by calling `app` from 2 and returning to 5. At point 6, 0CFA thinks that each variable can be bound to either 1 or 2. (For polyvariant analyses, we can create similar examples.) On the other hand, if we only consider paths that respect call/return matching, there is no spurious flow of data. At 6, `n1` and `n2` are bound to constants.

2.2 Stack-change calculation

Besides data-flow information, pushdown models also improve control-flow information. Hence, we can use them to accurately calculate stack changes between program points. With call/return matching, there is only one execution path in our example:

1 2 8 9 12 13 14 10 11 3 4 8 9 12 13 14 10 11 5 6 7

In contrast, 0CFA thinks that the program has a loop (there is a path from 4 to 3).

Many optimizations require accurate information about stack change. For instance:

- Most compound data are heap allocated in the general case. Examples include: closure environments, `cons` pairs, records, objects, *etc*. If we can show statically that such a piece of data is only passed downward, we can allocate it on the stack and reduce garbage-collection overhead.
- Continuations captured by `call/cc` may not escape upward. In this case, we do not need to copy the stack into the heap.
- In object-oriented languages, objects may have methods that are thread-safe by using locks. An escape analysis can eliminate unnecessary locking/unlocking in the methods of thread-private objects.

Such optimizations are better performed with pushdown models.

2.3 Fake rebinding

It is possible that two references to the same variable are always bound in the same runtime environment. If a flow analysis cannot detect that, it may allow paths in which the two references are bound to different abstract values. We call this phenomenon fake rebinding [21].

```
(define (compose-same f x) (f (f x)))
```

In `compose-same`, both references to `f` are always bound in the same environment (the top stack frame). However, if multiple closures flow to `f`, a finite-state analysis may call one closure at the inner call site and a different closure at the outer call site. CFA2 forbids this path because it knows that both references are bound in the top frame.

2.4 Broadening the applicability of pushdown models

Pushdown-reachability algorithms use a dynamic-programming technique called summarization. Summarization relies on proper nesting of calls and returns. If we call `app` from 2 in our example, summarization knows that, if the call returns, it will return to 3.

What if the call to `app` is a tail call, in which case the return point is in a different procedure from the call site? We can make this work by creating *cross-procedure* summaries [21].

In languages with exceptions, the return point may be deeper in the stack. We can transform this case into ordinary call/return nesting and handle it precisely with CFA2. Instead of thinking of an exception as a single jump deeper in the stack, we can return to the caller, which checks if it can handle the exception and if not, it passes it to its own caller and so on. Functions return a pair of values, one for normal return and one for exceptional return. The JavaScript implementation of CFA2 [1] uses this technique for exceptions.

But what if the return point has been popped off, as is the case when using first-class control constructs? Pushdown models cannot currently analyze such programs, so we have to fall back to a finite-state analysis and live with its limitations. In the rest of this paper, we show how to generalize pushdown models to first-class control.

$$
\begin{array}{rcl}
v \in Var & = & UVar + CVar \\
u \in UVar & = & \text{a set of identifiers} \\
k \in CVar & = & \text{a set of identifiers} \\
\psi \in Lab & = & ULab + CLab \\
l \in ULab & = & \text{a set of labels} \\
\gamma \in CLab & = & \text{a set of labels} \\
lam \in Lam & = & ULam + CLam \\
ulam \in ULam & ::= & [\![(\lambda_l\,(u\,k)\ call)]\!] \\
clam \in CLam & ::= & [\![(\lambda_\gamma\,(u)\ call)]\!] \\
call \in Call & = & UCall + CCall \\
UCall & ::= & [\![(f\ e\ q)^l]\!] \\
CCall & ::= & [\![(q\ e)^\gamma]\!] \\
g \in Exp & = & UExp + CExp \\
f, e \in UExp & = & ULam + UVar \\
q \in CExp & = & CLam + CVar \\
pr \in Program & ::= & ULam
\end{array}
$$

Figure 2: Partitioned CPS

3. Restricted CPS

Preliminary definitions In this section we describe our CPS language. Compilers that use CPS [5, 11, 20] usually partition the terms in a program in two disjoint sets, the user and the continuation set, and treat user terms differently from continuation terms. We adopt this partitioning here (Fig. 2). Variables, lambdas and calls get labels from $ULab$ or $CLab$. Labels are pairwise distinct. User lambdas take a user argument and the current continuation; continuation lambdas take only a user argument.

We assume that all variables in a program have distinct names. Then, the defining lambda of a variable v, written $def_\lambda(v)$, is the lambda term that contains v in its list of formals. For any term g, $iu_\lambda(g)$ is the innermost user lambda that contains g. Concrete syntax enclosed in $[\![\cdot]\!]$ denotes an item of abstract syntax. Functions with a '?' subscript are predicates, *e.g.*, $Var_?(e)$ returns true if e is a variable and false otherwise.

We use two notations for tuples, $(e_1, \ldots, e_n)$ and $\langle e_1, \ldots, e_n\rangle$, to avoid confusion when tuples are deeply nested. We use the latter for lists as well; ambiguities will be resolved by the context. Lists are also described by a head-tail notation, *e.g.*, $3 :: \langle 1, 3, -47\rangle$.

Handling first-class control In CPS, we can naturally express first-class control without using special primitives: when continuations are captured by user closures, they may escape.

Escaping continuations complicate reasoning about the stack. To permit effective reasoning about the stack in the presence of first-class control, we have previously proposed a syntactically-restricted variant of CPS, called Restricted CPS (*abbrev.* RCPS) [23].

Definition 1 (Restricted CPS). *A program is in Restricted CPS iff a continuation variable can appear free in a user lambda in operator position only.*

In RCPS, continuations escape in a well-behaved way: *a continuation can only be called after its escape,* it cannot be passed as an argument again. For example, the CPS-translation of `call/cc`, which is `(λ(f cc) (f (λ(v k) (cc v)) cc))`, is a valid RCPS term. Terms like `(λ(x k) (k (λ(y k2) (y 123 k))))` are not valid.

We can transform this term (and any CPS term) to a valid RCPS term by η-expanding to bring the free reference in operator position: `(λ(x k) (k (λ(y k2) (y 123 (λ(u) (k u))))))`. Why do we separate these very similar terms? Because, according to the Orbit policy (*cf.* sec. 4.1), their stack behaviors differ. In the case of the first term, when execution reaches `(y 123 k)`, we must restore the environment of the continuation that flows to `k`, which may cause arbitrary change to the stack. In the second case, when execution reaches `(y 123 (λ(u) (k u)))`, a new continuation is born and no stack change is required. Thus, RCPS forces all exotic stack change to happen when calling an escaping continuation, not in other kinds of call sites.

$$
\begin{array}{rcl}
\varsigma \in State & = & Eval + Apply \\
Eval & = & UEval + CEval \\
UEval & = & UCall \times BEnv \times VEnv \times Time \\
CEval & = & CCall \times BEnv \times VEnv \times Time \\
Apply & = & UApply + CApply \\
UApply & = & UClos \times UClos \times CClos \times VEnv \times Time \\
CApply & = & CClos \times UClos \times VEnv \times Time \\
Clos & = & UClos + CClos \\
d \in UClos & = & ULam \times BEnv \\
c \in CClos & = & (CLam \times BEnv) + halt \\
\beta \in BEnv & = & Var \rightharpoonup Time \\
ve \in VEnv & = & Var \times Time \rightharpoonup Clos \\
t \in Time & = & Lab^*
\end{array}
$$

(a) Concrete domains

$$
\mathcal{A}(g, \beta, ve) \triangleq \begin{cases} (g, \beta) & Lam_?(g) \\ ve(g, \beta(g)) & Var_?(g) \end{cases}
$$

[UEA] $([\![(f\ e\ q)^l]\!], \beta, ve, t) \to (proc, d, c, ve, l :: t)$
$proc = \mathcal{A}(f, \beta, ve)$
$d = \mathcal{A}(e, \beta, ve)$
$c = \mathcal{A}(q, \beta, ve)$

[UAE] $(proc, d, c, ve, t) \to (call, \beta', ve', t)$
$proc \equiv \langle [\![(\lambda_l\,(u\,k)\ call)]\!], \beta\rangle$
$\beta' = \beta[u \mapsto t][k \mapsto t]$
$ve' = ve[(u, t) \mapsto d][(k, t) \mapsto c]$

[CEA] $([\![(q\ e)^\gamma]\!], \beta, ve, t) \to (proc, d, ve, \gamma :: t)$
$proc = \mathcal{A}(q, \beta, ve)$
$d = \mathcal{A}(e, \beta, ve)$

[CAE] $(proc, d, ve, t) \to (call, \beta', ve', t)$
$proc = \langle [\![(\lambda_\gamma\,(u)\ call)]\!], \beta\rangle$
$\beta' = \beta[u \mapsto t]$
$ve' = ve[(u, t) \mapsto d]$

(b) Concrete semantics

Figure 3: Concrete semantics and domains

Concrete semantics Execution in RCPS is guided by the semantics of Fig. 3. In the terminology of abstract interpretation, this semantics is called the *concrete* semantics. In order to find properties of a program at compile time, one needs to derive a computable approximation of the concrete semantics, called the *abstract* semantics (*cf.* sec. 4).

Execution traces alternate between $Eval$ and $Apply$ states. At an $Eval$ state, we evaluate the subexpressions of a call site before performing a call. At an $Apply$, we perform the call.

The last component of each state is a *time*, which is a sequence of call sites. $Eval$-to-$Apply$ transitions increment the time by recording the label of the corresponding call site. $Apply$-to-$Eval$ transitions leave the time unchanged. Thus, the time t of a state reveals the call sites along the execution path to that state.

Times indicate points in the execution when variables are bound. The binding environment β is a partial function that maps variables to their binding times. The variable environment ve maps variable/time pairs to values. To find the value of a variable v, we look up the time v was put in β, and use that to search for the actual value in ve. By pairing variables with times, we allow a single variable to have multiple bindings at runtime.

Let's look at each transition individually. At a $UEval$ state over $[\![(f\ e\ q)^l]\!]$, we use the function $\mathcal{A}$ to evaluate the atomic expressions f, e and q. Lambdas are paired up with β to become closures, while variables are looked up in ve using β. We add the label l in front of the current time and transition to a $UApply$ state (rule [UEA]).

From $UApply$ to $Eval$, we bind the formals of a procedure $\langle[\![(\lambda_l\,(u\,k)\ call)]\!], \beta\rangle$ to the arguments and jump to its body. The new binding environment β' extends the procedure's environment, with u and k mapped to the current time. The new variable environment ve' maps (u, t) to the user argument d, and (k, t) to the continuation c (rule [UAE]).

The remaining two transitions are similar. We use $halt$ to denote the top-level continuation of a program pr. The initial state $\mathcal{I}(pr)$ is $((pr, \emptyset), input, halt, \emptyset, \langle\rangle)$, where $input$ is a closure of the form $\langle[\![(\lambda_l\,(u\,k)\ call)]\!], \emptyset\rangle$. The initial time is the empty sequence of call sites.

4. The CFA2 abstraction

This section shows how to extend the abstract semantics of CFA2 to handle first-class control. The semantics uses two binding environments, a stack and a heap. We also use the stack for return-point information.

We show the actual transition rules in section 4.3; the main difference from the previous semantics is that continuations can now be copied to, and restored from, the heap. Before that, we discuss how to manage the stack in RCPS (sec. 4.1) and how to decide whether a variable reference will be looked up in the stack or the heap (sec. 4.2). In section 4.4, we prove that the abstract semantics is an approximation of the concrete semantics.

The CFA2 abstraction is the first half of the story: the abstract state space is infinite, so we cannot explore it by enumerating all states. We tackle this problem in section 5.

4.1 Stack-management policy

The Orbit compiler [10, 11] compiles a CPS intermediate representation to final code that uses a stack. Orbit views continuations as closures whose environment record is a stack frame. To decide when to push and pop the stack, we follow Orbit's policy. The main idea behind Orbit's policy is that *we can manage the stack for a CPS program in the same way that we would manage it for the original direct-style program:*

- For every call to a user function, we push a frame for the arguments.
- We pop a frame at function returns. In CPS, user functions "return" by calling the current continuation with a return value.
- We also pop a frame at tail calls. A $UCall$ call site is a tail call in CPS iff it was a tail call in the original direct-style program. In tail calls, the continuation argument is a variable.
- When a continuation is captured by a user closure, we copy the stack into the heap.
- When we call a continuation that has escaped, we restore its stack from the heap.

4.2 Stack/heap split

The stack in CFA2 is more than a control structure for return-point information; it is also an *environment* structure—it contains bindings. CFA2 has a novel approach to variable binding: *two references to the same variable need not be looked up in the same binding environment.* We split references into two categories: stack and heap references. In direct-style, if a reference appears at the same nesting level as its binder, then it is a stack reference, otherwise it is a heap reference. For example, `(`λ_1`(x)(`λ_2`(y)(x (x y))))` has a stack reference to `y` and two heap references to `x`.

Intuitively, only heap references may escape. When we call a user function, we push a frame for its arguments, so we know that stack references are always bound in the top frame. When control reaches a heap reference, its frame is either deeper in the stack, or it has been popped. We look up stack references in the top frame, and heap references in the heap. Stack lookups below the top frame never happen (Fig. 4b).

When a program p is CPS-converted to a program p', stack (*resp.* heap) references in p remain stack (*resp.* heap) references in p'. All references added by the transform are stack references.

We can give an equivalent definition of stack and heap references directly in CPS, without referring to the original direct-style program. Labels can be split into disjoint sets according to the innermost user lambda that contains them. For the CPS translation of the previous program,

```
(λ1(x k1)
   (k1 (λ2(y k2)
          (x y (λ3(u) (x u k2)^4))^5))^6)
```

these sets are $\{1, 6\}$ and $\{2, 3, 4, 5\}$. The "label to variable" map $LV(\psi)$ returns all the variables bound by any lambdas that belong in the same set as ψ, *e.g.*, $LV(4) = \{\texttt{y}, \texttt{k2}, \texttt{u}\}$ and $LV(6) = \{\texttt{x}, \texttt{k1}\}$. We use this map to model stack behavior, because all continuation lambdas that "belong" to a given user lambda λ_l get closed by extending λ_l's stack frame (*cf.* section 4.3). Notice that, for any ψ, $LV(\psi)$ contains exactly one continuation variable. Using LV, we give the following definition.

Definition 2 (Stack and heap references)**.**

- *Let ψ be a call site that refers to a variable v. The predicate $S_?(\psi, v)$ holds iff $v \in LV(\psi)$. We call v a **stack reference**.*
- *Let ψ be a call site that refers to a variable v. The predicate $H_?(\psi, v)$ holds iff $v \notin LV(\psi)$. We call v a **heap reference**.*
- *v is a **stack variable**, written $S_?(v)$, iff all its references satisfy $S_?$.*
- *v is a **heap variable**, written $H_?(v)$, iff some of its references satisfy $H_?$.*

For instance, $S_?(5, \texttt{y})$ holds because $\texttt{y} \in \{\texttt{y}, \texttt{k2}, \texttt{u}\}$ and $H_?(5, \texttt{x})$ holds because $\texttt{x} \notin \{\texttt{y}, \texttt{k2}, \texttt{u}\}$.

4.3 Abstract semantics

The CFA2 semantics is an abstract machine that executes a program in RCPS (Fig. 4). The abstract domains appear in Fig. 4a. An abstract user closure (member of the set $\widehat{UClos}$) is a set of user lambdas. An abstract continuation closure (member of $\widehat{CClos}$) is either a continuation lambda or $halt$. A frame is a map from variables to abstract values, and a stack is a sequence of frames. All stack operations except $push$ are defined for non-empty stacks only. A heap is a map from variables to abstract values. In contrast to the previous semantics of CFA2, the heap can contain continuation bindings.

Fig. 4c shows the transition rules. First-class control shows up in two of the rules, $[\widehat{\text{UAE}}]$ and $[\widehat{\text{CEA}}]$.

On transition from a $\widehat{UEval}$ state to a $\widehat{UApply}$ state (rule $[\widehat{\text{UEA}}]$), we first evaluate f, e and q. We evaluate atomic user terms using $\hat{\mathcal{A}}_u$. We non-deterministically choose one of the lambdas that flow to f as the operator in the $\widehat{UApply}$ state.[1] The change

[1] An abstract execution explores one path, but the algorithm that searches the state space considers all possible executions.

$$\hat{\varsigma} \in \widehat{UEval} = UCall \times Stack \times Heap$$
$$\hat{\varsigma} \in \widehat{UApply} = ULam \times \widehat{UClos} \times \widehat{CClos} \times Stack \times Heap$$
$$\hat{\varsigma} \in \widehat{CEval} = CCall \times Stack \times Heap$$
$$\hat{\varsigma} \in \widehat{CApply} = \widehat{CClos} \times \widehat{UClos} \times Stack \times Heap$$
$$\hat{d} \in \widehat{UClos} = Pow(ULam)$$
$$\hat{c} \in \widehat{CClos} = CLam + halt$$
$$fr, tf \in Frame = (UVar \rightharpoonup \widehat{UClos}) + (CVar \rightharpoonup \widehat{CClos})$$
$$st \in Stack = Frame^*$$
$$h \in Heap = (UVar \rightharpoonup \widehat{UClos}) + (CVar \rightharpoonup Pow(\widehat{CClos} \times Stack))$$

(a) Abstract domains

$$\begin{aligned} pop(tf :: st) &\triangleq st \\ push(fr, st) &\triangleq fr :: st \\ (tf :: st)(v) &\triangleq tf(v) \\ (tf :: st)[u \mapsto \hat{d}] &\triangleq tf[u \mapsto \hat{d}] :: st \end{aligned}$$

(b) Stack operations

$$\hat{\mathcal{A}}_u(e, \psi, st, h) \triangleq \begin{cases} \{e\} & Lam_?(e) \\ st(e) & S_?(\psi, e) \\ h(e) & H_?(\psi, e) \end{cases}$$

$[\widehat{\text{UEA}}]$ $(\llbracket(\texttt{f e q})^l\rrbracket, st, h) \leadsto (ulam, \hat{d}, \hat{c}, st', h)$

$$ulam \in \hat{\mathcal{A}}_u(f, l, st, h)$$
$$\hat{d} = \hat{\mathcal{A}}_u(e, l, st, h)$$
$$\hat{c} = \begin{cases} st(q) & Var_?(q) \\ q & Lam_?(q) \end{cases}$$
$$st' = \begin{cases} pop(st) & Var_?(q) \\ st & Lam_?(q) \wedge (H_?(l, f) \vee Lam_?(f)) \\ st[f \mapsto \{ulam\}] & Lam_?(q) \wedge S_?(l, f) \end{cases}$$

$[\widehat{\text{UAE}}]$ $(\llbracket(\lambda_l\texttt{(u k)}\ call)\rrbracket, \hat{d}, \hat{c}, st, h) \leadsto (call, st', h')$

$$st' = push([u \mapsto \hat{d}][k \mapsto \hat{c}], st)$$
$$h'(v) = \begin{cases} h(u) \cup \hat{d} & (v = u) \wedge H_?(u) \\ h(k) \cup \{(\hat{c}, st)\} & (v = k) \wedge H_?(k) \\ h(v) & o/w \end{cases}$$

$[\widehat{\text{CEA}}]$ $(\llbracket(\texttt{q e})^\gamma\rrbracket, st, h) \leadsto (\hat{c}, \hat{d}, st', h)$

$$\hat{d} = \hat{\mathcal{A}}_u(e, \gamma, st, h)$$
$$(\hat{c}, st') \in \begin{cases} \{(q, st)\} & Lam_?(q) \\ \{(st(q), pop(st))\} & S_?(\gamma, q) \\ h(q) & H_?(\gamma, q) \end{cases}$$

$[\widehat{\text{CAE}}]$ $(\llbracket(\lambda_\gamma\texttt{(u)}\ call)\rrbracket, \hat{d}, st, h) \leadsto (call, st', h')$

$$st' = st[u \mapsto \hat{d}]$$
$$h'(v) = \begin{cases} h(u) \cup \hat{d} & (v = u) \wedge H_?(u) \\ h(v) & o/w \end{cases}$$

(c) Abstract semantics

Figure 4: Abstract semantics and relevant definitions

to the stack depends on q and f. If q is a variable, the call is a tail call so we pop the stack (case 1). If q is a lambda, it evaluates to a new closure whose environment is the top frame, hence we do not pop the stack (cases 2, 3). Moreover, if f is a lambda or a heap reference then we leave the stack unchanged. However, if f is a stack reference, we set f's value in the top frame to $\{ulam\}$, possibly forgetting other lambdas that flow to f. The strong update to the stack prevents fake rebinding for stack references (*cf.* sec. 2.3): when we return to $\hat{c}$, we may reach more stack references of f. These references and the current one are bound at the same time. Therefore, they must also be bound to $ulam$.

In the $\widehat{UApply}$-to-$\widehat{Eval}$ transition (rule $[\widehat{\text{UAE}}]$), we push a frame for the procedure's arguments. If u is a heap variable, we update its binding in the heap with all lambdas in $\hat{d}$. If k is a heap variable, we have a possibly escaping continuation. We save $\hat{c}$ in the heap and also copy the stack, so that we can restore it if $\hat{c}$ gets called later.

In a $\widehat{CEval}$-to-$\widehat{CApply}$ transition (rule $[\widehat{\text{CEA}}]$), we are preparing for a call to a continuation so we must reset the stack to the stack of its birth. When q is a lambda, it is a newly created closure so the stack does not change. When q is a stack reference, the $\widehat{CEval}$ state is a function return and the continuation's environment is the second stack frame. Therefore, we pop a frame before calling $\hat{c}$. When q is a heap reference, we are calling a continuation that may have escaped. The stack change since the continuation capture can be arbitrary. We non-deterministically pick a pair $(\hat{c}, st')$ from $h(q)$, jump to $\hat{c}$ and restore st', which contains bindings for the stack references in $\hat{c}$.

In the $\widehat{CApply}$-to-$\widehat{Eval}$ transition (rule $[\widehat{\text{CAE}}]$), the top frame is the environment of $\llbracket(\lambda_\gamma\texttt{(u)}\ call)\rrbracket$; stack references in $call$ need this frame on the top of the stack. Hence, we do not push; we extend the top frame with the binding for the continuation's parameter u. If u is a heap variable, we also update the heap.

Example Let's see how the abstract semantics works on a program with `call/cc`. Consider the program

$$\texttt{(call/cc (}\lambda\texttt{(c) (}somefun\texttt{ (c 42))))}$$

where $somefun$ is an arbitrary function. We use `call/cc` to capture the top-level continuation and bind it to `c`. Then, $somefun$ will never be called, because `(c 42)` will return to the top level with 42 as the result.

The CPS translation of `call/cc` is

$$\texttt{(}\lambda_1\texttt{(f cc) (f (}\lambda_2\texttt{(x k2) (cc x)) cc))}$$

The CPS translation of its argument is

$$\texttt{(}\lambda_3\texttt{(c k) (c 42 (}\lambda_4\texttt{(u) (}somefun_{\text{CPS}}\texttt{ u k))))}$$

The initial state $\hat{\mathcal{I}}(pr)$ is a $\widehat{UApply}$. We abbreviate lambdas by their labels.

$$(\lambda_1,\ \{\lambda_3\},\ halt,\ \langle\rangle,\ \emptyset)$$

We push a frame and jump to the body of λ_1. Since `cc` is a heap variable, we save the continuation and the stack in the heap. The heap h contains a single binding $[\texttt{cc} \mapsto \{(halt, \langle\rangle)\}]$.

$$(\llbracket\texttt{(f }\lambda_2\texttt{ cc)}\rrbracket,\ \langle[\texttt{f} \mapsto \{\lambda_3\}][\texttt{cc} \mapsto halt]\rangle,\ h)$$

λ_2 is essentially a continuation reified as a user value. We tail call to λ_3, so we pop the stack.

$$(\lambda_3,\ \{\lambda_2\},\ halt,\ \langle\rangle,\ h)$$

We push a frame and jump to the body of λ_3.

$$(\llbracket\texttt{(c 42 }\lambda_4\texttt{)}\rrbracket,\ \langle[\texttt{c} \mapsto \{\lambda_2\}][\texttt{k} \mapsto halt]\rangle,\ h)$$

This is a non-tail call, so we do not pop.

$$(\lambda_2,\ \{\texttt{42}\},\ \lambda_4,\ \langle[\texttt{c} \mapsto \{\lambda_2\}][\texttt{k} \mapsto halt]\rangle,\ h)$$

$$|(\llbracket(g_1 \ldots g_n)^{\psi}\rrbracket, \beta, ve, t)|_{ca} = (\llbracket(g_1 \ldots g_n)^{\psi}\rrbracket,\ \mathit{toStack}(LV(\psi), \beta, ve),\ |ve|_{ca})$$

$$|(\langle\llbracket(\lambda_l\,(u\,k)\ \mathit{call})\rrbracket, \beta\rangle, d, c, ve, t)|_{ca} = (\llbracket(\lambda_l\,(u\,k)\ \mathit{call})\rrbracket, |d|_{ca}, |c|_{ca}, st, |ve|_{ca})$$
$$\text{where}\ \ st = \begin{cases} \langle\rangle & c = \mathit{halt} \\ \mathit{toStack}(LV(\gamma), \beta', ve) & c = (\llbracket(\lambda_\gamma(u')\mathit{call}')\rrbracket, \beta') \end{cases}$$

$$|(\langle\llbracket(\lambda_\gamma\,(u)\ \mathit{call})\rrbracket, \beta\rangle, d, ve, t)|_{ca} = (\llbracket(\lambda_\gamma\,(u)\ \mathit{call})\rrbracket,\ |d|_{ca}, \mathit{toStack}(LV(\gamma), \beta, ve),\ |ve|_{ca})$$

$$|(\mathit{halt}, d, ve, t)|_{ca} = (\mathit{halt}, |d|_{ca},\ \langle\rangle,\ |ve|_{ca})$$

$$|(\llbracket(\lambda_l\,(u\,k)\ \mathit{call})\rrbracket, \beta)|_{ca} = \{\llbracket(\lambda_l\,(u\,k)\ \mathit{call})\rrbracket\}$$

$$|(\llbracket(\lambda_\gamma\,(u)\ \mathit{call})\rrbracket, \beta)|_{ca} = \llbracket(\lambda_\gamma\,(u)\ \mathit{call})\rrbracket$$

$$|\mathit{halt}|_{ca} = \mathit{halt}$$

$$|ve|_{ca} = \{\ (u, \textstyle\bigcup_t |ve(u,t)|_{ca})\ :\ (u \in \mathit{UVar}) \wedge H_?(u)\} \cup \{\ (k, \textstyle\bigcup_t \mathit{makecs}(ve(k,t), ve))\ :\ (k \in \mathit{CVar}) \wedge H_?(k)\}$$
$$\text{where}\ \ \mathit{makecs}(c, ve) \triangleq \begin{cases} (\mathit{halt}, \langle\rangle) & c = \mathit{halt} \\ (\llbracket(\lambda_\gamma(u')\mathit{call})\rrbracket, \mathit{toStack}(LV(\gamma), \beta, ve)) & c = \langle\llbracket(\lambda_\gamma(u')\mathit{call})\rrbracket, \beta\rangle \end{cases}$$

$$\mathit{toStack}(\{u_1, \ldots, u_n, k\}, \beta, ve) \triangleq \begin{cases} \langle[\overline{u_i \mapsto \hat{d}_i}\,][k \mapsto \mathit{halt}]\rangle & ve(k, \beta(k)) = \mathit{halt} \\ [\overline{u_i \mapsto \hat{d}_i}\,][k \mapsto \llbracket(\lambda_\gamma\,(u)\ \mathit{call})\rrbracket] :: st & ve(k, \beta(k)) = (\llbracket(\lambda_\gamma\,(u)\ \mathit{call})\rrbracket, \beta') \end{cases}$$
$$\text{where}\ \ \hat{d}_i = |ve(u_i, \beta(u_i))|_{ca}\ \ \text{and}\ \ st = \mathit{toStack}(LV(\gamma), \beta', ve)$$

Figure 5: From concrete states to abstract states

$$g \sqsubseteq g \quad \text{where} \quad g \in (\mathit{halt} + \mathit{Lam} + \mathit{Call})$$

$$\langle a_1, \ldots, a_n\rangle \sqsubseteq \langle b_1, \ldots, b_n\rangle \quad \text{iff} \quad \text{for } 1 \leqslant i \leqslant n,\ a_i \sqsubseteq b_i$$

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

$$h_1 \sqsubseteq h_2 \quad \text{iff} \quad h_1(v) \sqsubseteq h_2(v) \quad \text{for each } v \in \mathrm{dom}(h_1)$$

$$\mathit{tf}_1 :: st_1 \sqsubseteq \mathit{tf}_2 :: st_2 \quad \text{iff} \quad \mathit{tf}_1 \sqsubseteq \mathit{tf}_2 \ \wedge\ st_1 \sqsubseteq st_2$$

$$\langle\rangle \sqsubseteq \langle\rangle$$

$$\mathit{tf}_1 \sqsubseteq \mathit{tf}_2 \quad \text{iff} \quad \mathit{tf}_1(v) \sqsubseteq \mathit{tf}_2(v) \quad \text{for each } v \in \mathrm{dom}(\mathit{tf}_1)$$

Figure 6: The $\sqsubseteq$ relation on abstract states

We push a frame and jump to the body of λ_2.

$$(\llbracket(\mathtt{cc\ x})\rrbracket,\ \langle[\mathtt{x} \mapsto \{\mathtt{42}\}][\mathtt{k2} \mapsto \lambda_4],\ [\mathtt{c} \mapsto \{\lambda_2\}][\mathtt{k} \mapsto \mathit{halt}]\rangle,\ h)$$

`cc` is a heap reference, so we ignore the current continuation and stack and restore $(\mathit{halt}, \langle\rangle)$ from the heap.

$$(\mathit{halt},\ \{\mathtt{42}\},\ \langle\rangle,\ h)$$

The program terminates with value $\{\mathtt{42}\}$.

4.4 Correctness of the abstract semantics

In this section, we show that the abstract semantics *simulates* the concrete semantics, which means that the execution of a program under the abstract semantics is a safe approximation of its actual runtime behavior. First, we define a map $|\cdot|_{ca}$ from concrete to abstract states. Next, we show that if ς transitions to ς' in the concrete semantics, the abstract counterpart $|\varsigma|_{ca}$ of ς transitions to a state $\hat{\varsigma}'$ which approximates $|\varsigma'|_{ca}$. Therefore, each concrete execution, *i.e.*, sequence of states related by $\rightarrow$, has a corresponding abstract execution that computes an approximate answer.

The map $|\cdot|_{ca}$ appears in Fig. 5. The abstraction of an Eval state ς of the form $(\llbracket(g_1 \ldots g_n)^{\psi}\rrbracket, \beta, ve, t)$ is an $\widehat{\mathit{Eval}}$ state $\hat{\varsigma}$ with the same call site. Since ς does not have a stack, we must expose stack-related information hidden in β and ve. Assume that λ_l is the innermost user lambda that contains ψ. To reach ψ, control passed from a $\widehat{\mathit{UApply}}$ state $\hat{\varsigma}'$ over λ_l. According to our stack policy, the top frame contains bindings for the formals of λ_l and any temporaries added along the path from $\hat{\varsigma}'$ to $\hat{\varsigma}$. Therefore, the domain of the top frame is a subset of $LV(l)$, *i.e.*, a subset of $LV(\psi)$. For each user variable $u_i \in (LV(\psi) \cap \mathrm{dom}(\beta))$, the top frame contains $[u_i \mapsto |ve(u_i, \beta(u_i))|_{ca}]$. Let k be the sole continuation variable in $LV(\psi)$. If $ve(k, \beta(k))$ is halt (the return continuation is the top-level continuation), the rest of the stack is empty. If $ve(k, \beta(k))$ is $(\llbracket(\lambda_\gamma\,(u)\ \mathit{call})\rrbracket, \beta')$, the second frame is for the user lambda in which λ_γ was born, and so forth: proceeding through the stack, we add a frame for each live activation of a user lambda until we reach halt.

The abstraction of a UApply state over $\langle\llbracket(\lambda_l\,(u\,k)\ \mathit{call})\rrbracket, \beta\rangle$ is a $\widehat{\mathit{UApply}}$ state $\hat{\varsigma}$ whose operator is $\llbracket(\lambda_l\,(u\,k)\ \mathit{call})\rrbracket$. The stack of $\hat{\varsigma}$ represents the environment in which the continuation argument was created, and we compute it using $\mathit{toStack}$ as above.

Abstracting a CApply is similar to the UApply case, only now the top frame is the environment of the continuation operator. Note that the abstraction maps drop the time of the concrete states, since the abstract states do not use times.

The abstraction of a user closure is the singleton set with the corresponding lambda. The abstraction of a continuation closure is the corresponding lambda.

The abstraction $|ve|_{ca}$ of a variable environment is a heap, which contains bindings for the user and the continuation heap variables. Each heap user variable is bound to the set of lambdas of the closures that can flow to it. Each heap continuation variable k is bound to a set of continuation-stack pairs. For each closure that can flow to k, we create a pair with the lambda of that closure and the corresponding stack.

The relation $\hat{\varsigma}_1 \sqsubseteq \hat{\varsigma}_2$ is a partial order on abstract states and can be read as "$\hat{\varsigma}_1$ is more precise than $\hat{\varsigma}_2$" (Fig. 6). Tuples are ordered pointwise. Abstract user closures are ordered by inclusion. Two stacks are in $\sqsubseteq$ iff they have the same length and the corresponding frames are in $\sqsubseteq$.

We can now state the simulation theorem. The proof proceeds by case analysis on the concrete transition relation. It can be found in the appendix.

Theorem 3 (Simulation). *If $\varsigma \to \varsigma'$ and $|\varsigma|_{ca} \sqsubseteq \hat{\varsigma}$, then there exists $\hat{\varsigma}'$ such that $\hat{\varsigma} \rightsquigarrow \hat{\varsigma}'$ and $|\varsigma'|_{ca} \sqsubseteq \hat{\varsigma}'$.*

5. Exploring the infinite state space

Pushdown-reachability algorithms, including CFA2, deal with the unbounded stack size by using a dynamic-programming technique called *summarization*. These algorithms work on transition systems whose stack is unbounded, but the rest of the components are bounded. Due to escaping continuations, we also have to deal with infinitely-many heaps.

5.1 Overview of summarization

We start with an informal overview of summarization. Assume that a program is executing and control reaches the entry of a procedure. We start computing inside the procedure. While doing so, we are visiting several program points inside the procedure and possibly calling (and returning from) other procedures. Sometime later, we reach the exit and are about to return to the caller with a result. The intuition behind summarization is that, during this computation, the return point was irrelevant; it influences reachability only after we return to the caller. Consequently, if from a program point n with an empty stack we can reach a point n' with stack s', then from n with s we can reach n' with *append*(s', s).

Let's use summarization to find which nodes of the graph of Fig. 1 are reachable from node 1. We find reachable nodes by recording *path edges*, *i.e.*, edges whose source is the entry of a procedure and target is some program point in the same procedure. Path edges should not be confused with the edges already present in the graph. They are artificial edges used by the analysis to represent intraprocedural paths, hence the name.

Node 1 goes to 2, so we record the edges $\langle 1, 1\rangle$ and $\langle 1, 2\rangle$. From 2 we call `app`, so we record the call $\langle 2, 8\rangle$ and jump to 8. In `app`, we find path edges $\langle 8, 8\rangle$ and $\langle 8, 9\rangle$. We find a new call $\langle 9, 12\rangle$ and jump to 12. Inside `id`, we discover the edges $\langle 12, 12\rangle$, $\langle 12, 13\rangle$ and $\langle 12, 14\rangle$. Edges that go from an entry to an exit, such as $\langle 12, 14\rangle$, are called *summary* edges. We have not been keeping track of the stack, so we use the recorded calls to find the return point. The only call to `id` is $\langle 9, 12\rangle$, so 14 returns to 10 and we find a new edge $\langle 8, 10\rangle$, which leads to $\langle 8, 11\rangle$. We record $\langle 8, 11\rangle$ as a summary also. From the call $\langle 2, 8\rangle$, we see that 11 returns to 3, so we record edges $\langle 1, 3\rangle$ and $\langle 1, 4\rangle$. Now, we have a new call to `app`. Reachability inside `app` does not depend on its calling context. From the summary $\langle 8, 11\rangle$, we know that 4 can reach 5, so we find $\langle 1, 5\rangle$. Subsequently, we find the last two path edges, which are $\langle 1, 6\rangle$ and $\langle 1, 7\rangle$.

During the search, we did two kinds of transitions. The first kind includes intraprocedural steps and calls; these transitions do not shrink the stack. The second is function returns, which shrink the stack. Since we are not keeping track of the stack, we find the target nodes of the second kind of transitions in an indirect way, by recording calls and summaries. We show a summarization-based algorithm for CFA2 in section 5.3. The next section describes the local semantics, which we use in the algorithm for transitions that do not shrink the stack.

$$
\begin{aligned}
\widetilde{Eval} &= Call \times \widetilde{Stack} \times \widetilde{Heap}\\
\widetilde{UApply} &= ULam \times \widetilde{UClos} \times \widetilde{Heap}\\
\widetilde{CApply} &= \widetilde{CClos} \times \widetilde{UClos} \times \widetilde{Stack} \times \widetilde{Heap}\\
\widetilde{Frame} &= UVar \rightharpoonup \widetilde{UClos}\\
\widetilde{Stack} &= \widetilde{Frame}\\
\widetilde{Heap} &= UVar \rightharpoonup \widetilde{UClos}
\end{aligned}
$$

(a) Local domains

$$
\begin{aligned}
&|(call, st, h)|_{al} = (call, |st|_{al}, |h|_{al})\\
&|(ulam, \hat{d}, \hat{c}, st, h)|_{al} = (ulam, \hat{d}, |h|_{al})\\
&|(\hat{c}, \hat{d}, st, h)|_{al} = (\hat{c}, \hat{d}, |st|_{al}, |h|_{al})\\
&|st|_{al} = \begin{cases} \emptyset & st = \langle\rangle \\ tf \restriction UVar & st = tf :: st' \end{cases}\\
&|h|_{al} = h \restriction UVar
\end{aligned}
$$

(b) Abstract to local maps

$$
\tilde{\mathcal{A}}_u(e, \psi, tf, h) \triangleq \begin{cases} \{e\} & Lam_?(e) \\ tf(e) & S_?(\psi, e) \\ h(e) & H_?(\psi, e) \end{cases}
$$

$$
\begin{aligned}
&[\widetilde{\text{UEA}}] \quad ([\![(f\ e\ q)^l]\!], tf, h) \approx\!> (ulam, \hat{d}, h)\\
&\qquad ulam \in \tilde{\mathcal{A}}_u(f, l, tf, h)\\
&\qquad \hat{d} = \tilde{\mathcal{A}}_u(e, l, tf, h)
\end{aligned}
$$

$$
\begin{aligned}
&[\widetilde{\text{UAE}}] \quad ([\![(\lambda_l (u\, k)\ call)]\!], \hat{d}, h) \approx\!> (call, [u \mapsto \hat{d}], h')\\
&\qquad h'(v) = \begin{cases} h(u) \cup \hat{d} & (v = u) \wedge H_?(u) \\ h(v) & o/w \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\widetilde{\text{CEA}}] \quad ([\![(clam\ e)^\gamma]\!], tf, h) \approx\!> (clam, \hat{d}, tf, h)\\
&\qquad \hat{d} = \tilde{\mathcal{A}}_u(e, \gamma, tf, h)
\end{aligned}
$$

$$
\begin{aligned}
&[\widetilde{\text{CAE}}] \quad ([\![(\lambda_\gamma (u)\ call)]\!], \hat{d}, tf, h) \approx\!> (call, tf', h')\\
&\qquad tf' = tf[u \mapsto \hat{d}]\\
&\qquad h'(v) = \begin{cases} h(u) \cup \hat{d} & (v = u) \wedge H_?(u) \\ h(v) & o/w \end{cases}
\end{aligned}
$$

(c) Local semantics

Figure 7: Local semantics and relevant definitions

5.2 Local semantics

Summarization operates on a finite set of program points. Since the abstract state space is infinite, we cannot use abstract states as program points. For this reason, we introduce *local states* (Fig. 7a) and define a map $|\cdot|_{al}$ from abstract to local states (Fig. 7b).

The local semantics (Fig. 7) describes executions that do not touch the rest of the stack (*i.e.*, executions where functions do not return). A $\widetilde{CEval}$ state with call site $[\![(k\ e)^\gamma]\!]$ has no successor in this semantics. Since functions do not call their continuations, the local frames and heaps contain only user bindings. Local steps are otherwise similar to abstract steps. Note that there is no provision for first-class control in the local transitions; they are identical to the previous ones [21]. The metavariable $\tilde{\varsigma}$ ranges over local states. We define the map $|\cdot|_{cl}$ from concrete to local states to be $|\cdot|_{al} \circ |\cdot|_{ca}$.

Summarization distinguishes between different kinds of states: entries, exits, calls, returns and inner states. CPS lends itself naturally to such a categorization. The following definition works for all three state spaces: concrete, abstract and local.

Definition 4 (Classification of states)**.**

- *A $UApply$ state is an **Entry**—control is about to enter the body of a function.*
- *A $CEval$ state is an **Exit-Ret** when the operator is a stack reference—a function is about to return a result to its context.*
- *A $CEval$ state is an **Exit-Esc** when the operator is a heap reference—we are calling a continuation that may have escaped.*
- *A $CEval$ state where the operator is a lambda is an **Inner** state.*
- *A $UEval$ state is an **Exit-TC** when the continuation argument is a variable—at tail calls control does not return to the caller.*
- *A $UEval$ state is a **Call** when the continuation argument is a lambda.*
- *A $CApply$ state is a **Return** if its predecessor is an exit, or an **Inner** state if its predecessor is also an inner state. Our algorithm does not distinguish between the two kinds of $CApply$s; the difference is just conceptual.*

5.3 The CFA2 algorithm

The algorithm for CFA2 appears in Fig. 8. Its goal is to compute which local states are reachable from the initial state of a program.

For readers familiar with the previous algorithm: the main addition is the handling of Exit-Esc states (lines 25-33). Escaping continuations also require changes to the handling of entries and tail calls. Entries are now a separate case, instead of together with $\widetilde{CApply}$s and inner $\widetilde{CEval}$s. Last, the `Propagate` function takes an extra argument.

Structure of the algorithm The algorithm uses a workset W, which contains path edges and summaries to be examined. An edge $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ is an ordered pair of local states. We call $\tilde{\varsigma}_1$ the *source* and $\tilde{\varsigma}_2$ the *target* of the edge. At every iteration, we remove an edge from W and process it, potentially adding new edges in W. We stop when W is empty.

An edge $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ is a summary when $\tilde{\varsigma}_1$ is an entry and $\tilde{\varsigma}_2$ is either an Exit-Ret or an Exit-Esc, not necessarily in the same procedure. Summaries carry an important message: *each continuation that can be passed to $\tilde{\varsigma}_1$ can flow to the operator position of $\tilde{\varsigma}_2$.*

The algorithm maintains several sets. The results of the analysis are stored in the set $Seen$. It contains path edges (from a procedure entry to a state in the same procedure) and summary edges. The target of an edge in $Seen$ is reachable from the source and from the initial state. Summaries are also stored in $Summary$. $Escapes$ contains Exit-Esc states. If the continuation parameter of a user lambda is a heap variable, entries over that lambda are stored in $EntriesEsc$. $Final$ records final states, *i.e.*, $\widetilde{CApply}$s that call $halt$ with a return value for the whole program. $Callers$ contains triples $\langle \tilde{\varsigma}_1, \tilde{\varsigma}_2, \tilde{\varsigma}_3 \rangle$, where $\tilde{\varsigma}_1$ is an entry, $\tilde{\varsigma}_2$ is a call in the same procedure and $\tilde{\varsigma}_3$ is the entry of the callee. $TCallers$ contains triples $\langle \tilde{\varsigma}_1, \tilde{\varsigma}_2, \tilde{\varsigma}_3 \rangle$, where $\tilde{\varsigma}_1$ is an entry, $\tilde{\varsigma}_2$ is a tail call in the same procedure and $\tilde{\varsigma}_3$ is the entry of the callee. The initial state $\widetilde{\mathcal{I}}(pr)$ is defined as $|\mathcal{I}(pr)|_{cl}$. The helper function $succ(\tilde{\varsigma})$ returns the successor(s) of $\tilde{\varsigma}$ according to the local semantics.

Summaries for first-class continuations Perhaps surprisingly, even though continuations can escape to the heap in the abstract semantics, we do not need continuations in the local heap. We can handle escaping continuations with summaries. Consider the example from section 4.3. When control reaches $[\![$`(cc x)`$]\!]$, we want to find which continuation flows to `cc`. We know that $def_\lambda($`cc`$)$ is λ_1. By looking at the single $\widehat{UApply}$ over λ_1, we find that $halt$ flows to `cc`. This suggests that, for escaping continuations, we need summaries of the form $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ where $\tilde{\varsigma}_2$ is an Exit-Esc over a call site $[\![(k\ e)^\gamma]\!]$ and $\tilde{\varsigma}_1$ is an entry over $def_\lambda(k)$.

Edge processing Each edge $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ is processed in one of six ways, depending on $\tilde{\varsigma}_2$. If $\tilde{\varsigma}_2$ is a return or an inner state (line 12), then its successor $\tilde{\varsigma}_3$ is a state in the same procedure. Since $\tilde{\varsigma}_2$ is reachable from $\tilde{\varsigma}_1$, $\tilde{\varsigma}_3$ is also reachable from $\tilde{\varsigma}_1$. If we have not already recorded the edge $(\tilde{\varsigma}_1, \tilde{\varsigma}_3)$, we do it now (line 44).

If $\tilde{\varsigma}_2$ is a call (line 14) then $\tilde{\varsigma}_3$ is the entry of the callee, so we propagate $(\tilde{\varsigma}_3, \tilde{\varsigma}_3)$ instead of $(\tilde{\varsigma}_1, \tilde{\varsigma}_3)$ (line 16). Also, we record the call in $Callers$. If an exit $\tilde{\varsigma}_4$ is reachable from $\tilde{\varsigma}_3$, it should return to the continuation born at $\tilde{\varsigma}_2$ (line 18). The function `Update` is responsible for computing the return state. We find the return value $\hat{d}$ by evaluating the expression e_4 passed to the continuation (lines 48-49). Since we are returning to λ_{γ_2}, we must restore the environment of its creation, which is tf_2 (possibly with stack filtering, line 50). The new state $\tilde{\varsigma}$ is the corresponding return of $\tilde{\varsigma}_2$, so we propagate $(\tilde{\varsigma}_1, \tilde{\varsigma})$ (lines 51-52).

If $\tilde{\varsigma}_2$ is an Exit-Ret and $\tilde{\varsigma}_1$ is the initial state (lines 19-20), then $\tilde{\varsigma}_2$'s successor is a final state (lines 53-54). If $\tilde{\varsigma}_1$ is some other entry, we record the edge in $Summary$ and pass the result of $\tilde{\varsigma}_2$ to the callers of $\tilde{\varsigma}_1$ (lines 22-23). Last, consider the case of a tail call $\tilde{\varsigma}_4$ to $\tilde{\varsigma}_1$ (line 24). No continuation is born at $\tilde{\varsigma}_4$. Thus, we must find where $\tilde{\varsigma}_3$ (the entry that led to the tail call) was called from. Then again, all calls to $\tilde{\varsigma}_3$ may be tail calls, in which case we keep searching further back in the call chain to find a return point. We do the backward search by transitively adding a cross-procedure summary from $\tilde{\varsigma}_3$ to $\tilde{\varsigma}_2$.

Let $\tilde{\varsigma}_2$ be an Exit-Esc over a call site $[\![(k\ e)^\gamma]\!]$ (line 25). Its predecessor $\tilde{\varsigma}'$ is an entry or a $\widetilde{CApply}$. To reach $\tilde{\varsigma}_2$, the algorithm must go through $\tilde{\varsigma}'$. Hence, the first time the algorithm sees $\tilde{\varsigma}_2$ is at line 7 or 13, which means that $\tilde{\varsigma}_1$ is an entry over $iu_\lambda([\![(k\ e)^\gamma]\!])$ and $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ is not in $Summary$. Thus, the test at line 26 is true. We record $\tilde{\varsigma}_2$ in $Escapes$. We also create summaries from entries over $def_\lambda(k)$ to $\tilde{\varsigma}_2$, in order to find which continuations can flow to k. We make sure to put these summaries in $Summary$ (line 29), so that when they are examined, the test at line 26 is false.

When $\tilde{\varsigma}_2$ is examined again, this time $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ is in $Summary$. If $\tilde{\varsigma}_1$ is the initial state, $\tilde{\varsigma}_2$ can call $halt$ and transition to a final state (line 30). Otherwise, we look for calls to $\tilde{\varsigma}_1$ to find continuations that can be called at $\tilde{\varsigma}_2$ (line 32). If there are tail calls to $\tilde{\varsigma}_1$, we propagate summaries transitively (line 33).

If $\tilde{\varsigma}_2$ is an entry over $[\![(\lambda_l\,(u\,k)\ call)]\!]$, its successor $\tilde{\varsigma}_3$ is a state in the same procedure, so we propagate $(\tilde{\varsigma}_1, \tilde{\varsigma}_3)$ (lines 6-7). If k is a heap variable (lines 8-9), we put $\tilde{\varsigma}_2$ in $EntriesEsc$ (so that it can be found from line 29). Also, if we have seen Exit-Esc states that call k, we create summaries from $\tilde{\varsigma}_2$ to those states (line 11).

If $\tilde{\varsigma}_2$ is a tail call (line 34), we find its successors and record the call in $TCallers$ (lines 35-37). If a successor of $\tilde{\varsigma}_2$ goes to an exit, we propagate a cross-procedure summary transitively (line 41). Moreover, if $\tilde{\varsigma}_4$ is an Exit-Esc, we want to make sure that $(\tilde{\varsigma}_1, \tilde{\varsigma}_4)$ is in $Summary$ when it is examined. We cannot call `Propagate` with `true` at line 41 because we would be mutating $Summary$ while iterating over it. Instead, we use a temporary set which we unite with $Summary$ after the loop (line 42).

```
$Summary$, $Callers$, $TCallers$, $EntriesEsc$, $Escapes$, $Final$ $\longleftarrow$ $\emptyset$
$Seen$, $W$ $\longleftarrow$ $\{(\tilde{\mathcal{I}}(pr),\ \tilde{\mathcal{I}}(pr))\}$
while $W \neq \emptyset$
  remove ($\tilde\varsigma_1$, $\tilde\varsigma_2$) from $W$
  switch $\tilde\varsigma_2$
    case $\tilde\varsigma_2$ of Entry
      for the $\tilde\varsigma_3$ in $succ(\tilde\varsigma_2)$, Propagate($\tilde\varsigma_1$, $\tilde\varsigma_3$, false)
      $\tilde\varsigma_2$ of the form $([\![(\lambda_l\,(u\,k)\ call)]\!],\ \hat{d},\ h)$
      if $H_?(k)$ then
        insert $\tilde\varsigma_2$ in $EntriesEsc$
        for each $\tilde\varsigma_3$ in $Escapes$ that calls $k$, Propagate($\tilde\varsigma_2$, $\tilde\varsigma_3$, true)
    case $\tilde\varsigma_2$ of CApply, Inner-CEval
      for the $\tilde\varsigma_3$ in $succ(\tilde\varsigma_2)$, Propagate($\tilde\varsigma_1$, $\tilde\varsigma_3$, false)
    case $\tilde\varsigma_2$ of Call
      for each $\tilde\varsigma_3$ in $succ(\tilde\varsigma_2)$
        Propagate($\tilde\varsigma_3$, $\tilde\varsigma_3$, false)
        insert ($\tilde\varsigma_1$, $\tilde\varsigma_2$, $\tilde\varsigma_3$) in $Callers$
        for each ($\tilde\varsigma_3$, $\tilde\varsigma_4$) in $Summary$, Update($\tilde\varsigma_1$, $\tilde\varsigma_2$, $\tilde\varsigma_3$, $\tilde\varsigma_4$)
    case $\tilde\varsigma_2$ of Exit-Ret
      if $\tilde\varsigma_1$ = $\tilde{\mathcal{I}}(pr)$ then Final($\tilde\varsigma_2$)
      else
        insert ($\tilde\varsigma_1$, $\tilde\varsigma_2$) in $Summary$
        for each ($\tilde\varsigma_3$, $\tilde\varsigma_4$, $\tilde\varsigma_1$) in $Callers$, Update($\tilde\varsigma_3$, $\tilde\varsigma_4$, $\tilde\varsigma_1$, $\tilde\varsigma_2$)
        for each ($\tilde\varsigma_3$, $\tilde\varsigma_4$, $\tilde\varsigma_1$) in $TCallers$, Propagate($\tilde\varsigma_3$, $\tilde\varsigma_2$, false)
    case $\tilde\varsigma_2$ of Exit-Esc
      if ($\tilde\varsigma_1$, $\tilde\varsigma_2$) not in $Summary$ then
        insert $\tilde\varsigma_2$ in $Escapes$
        $\tilde\varsigma_2$ of the form $([\![(k\ e)^{\gamma}]\!],\ tf,\ h)$
        for each $\tilde\varsigma_3$ in $EntriesEsc$ over $def_\lambda(k)$, Propagate($\tilde\varsigma_3$, $\tilde\varsigma_2$, true)
      else if $\tilde\varsigma_1$ = $\tilde{\mathcal{I}}(pr)$ then Final($\tilde\varsigma_2$)
      else
        for each ($\tilde\varsigma_3$, $\tilde\varsigma_4$, $\tilde\varsigma_1$) in $Callers$, Update($\tilde\varsigma_3$, $\tilde\varsigma_4$, $\tilde\varsigma_1$, $\tilde\varsigma_2$)
        for each ($\tilde\varsigma_3$, $\tilde\varsigma_4$, $\tilde\varsigma_1$) in $TCallers$, Propagate($\tilde\varsigma_3$, $\tilde\varsigma_2$, true)
    case $\tilde\varsigma_2$ of Exit-TC
      for each $\tilde\varsigma_3$ in $succ(\tilde\varsigma_2)$
        Propagate($\tilde\varsigma_3$, $\tilde\varsigma_3$, false)
        insert ($\tilde\varsigma_1$, $\tilde\varsigma_2$, $\tilde\varsigma_3$) in $TCallers$
        S $\longleftarrow$ $\emptyset$
        for each ($\tilde\varsigma_3$, $\tilde\varsigma_4$) in $Summary$
          insert ($\tilde\varsigma_1$, $\tilde\varsigma_4$) in S
          Propagate($\tilde\varsigma_1$, $\tilde\varsigma_4$, false)
        $Summary$ $\longleftarrow$ $Summary$ $\cup$ S

Propagate($\tilde\varsigma_1$, $\tilde\varsigma_2$, esc) $\triangleq$
  if esc then insert ($\tilde\varsigma_1$, $\tilde\varsigma_2$) in $Summary$
  if ($\tilde\varsigma_1$, $\tilde\varsigma_2$) not in $Seen$ then insert ($\tilde\varsigma_1$, $\tilde\varsigma_2$) in $Seen$ and $W$

Update($\tilde\varsigma_1$, $\tilde\varsigma_2$, $\tilde\varsigma_3$, $\tilde\varsigma_4$) $\triangleq$
  $\tilde\varsigma_1$ of the form $([\![(\lambda_{l_1}(u_1\ k_1)\ call_1)]\!],\ \hat{d}_1,\ h_1)$
  $\tilde\varsigma_2$ of the form $([\![(f\ e_2\ (\lambda_{\gamma_2}\ (u_2)\ call_2))^{l_2}]\!],\ tf_2,\ h_2)$
  $\tilde\varsigma_3$ of the form $([\![(\lambda_{l_3}(u_3\ k_3)\ call_3)]\!],\ \hat{d}_3,\ h_2)$
  $\tilde\varsigma_4$ of the form $([\![(k_4\ e_4)^{\gamma_4}]\!],\ tf_4,\ h_4)$
  $\hat{d} \longleftarrow \tilde{\mathcal{A}}_u(e_4,\ \gamma_4,\ tf_4,\ h_4)$
  $tf \longleftarrow \begin{cases} tf_2[f \mapsto \{[\![(\lambda_{l_3}(u_3\ k_3)\ call_3)]\!]\}] & S_?(l_2, f) \\ tf_2 & H_?(l_2, f) \vee Lam_?(f) \end{cases}$
  $\tilde\varsigma \longleftarrow ([\![(\lambda_{\gamma_2}(u_2)\ call_2)]\!],\ \hat{d},\ tf,\ h_4)$
  Propagate($\tilde\varsigma_1$, $\tilde\varsigma$, false)

Final($\tilde\varsigma$) $\triangleq$
  $\tilde\varsigma$ of the form $([\![(k\ e)^{\gamma}]\!],\ tf,\ h)$
  insert $(halt,\ \tilde{\mathcal{A}}_u(e,\ \gamma,\ tf,\ h),\ \emptyset,\ h)$ in $Final$
```

Figure 8: CFA2 workset algorithm

5.4 Soundness

The local state space is finite, so there are finitely many path and summary edges. We record edges as seen when we insert them in W, which ensures that no edge is inserted in W twice. Therefore, the algorithm always terminates.

We obviously cannot visit an infinite number of abstract states. To establish soundness, we relate the results of the algorithm to the abstract semantics: we show that if a state $\hat{\varsigma}$ is reachable from $\hat{\mathcal{I}}(pr)$, then the algorithm visits $|\hat{\varsigma}|_{al}$ (*cf.* theorem 8).

First-class continuations create an intricate call/return structure, which complicates reasoning about soundness. When calls and returns nest properly, an execution path can be decomposed so that for each state $\hat{\varsigma}$, we can uniquely identify another state $\hat{\varsigma}'$ as the entry of the procedure that contains $\hat{\varsigma}$ [16]. When we add tail calls into the mix, unique decomposition is still possible [22].

However, in the presence of first-class control, a state can belong to *more than one* procedure. For instance, suppose we want to find the entry of the procedure containing $\hat{\varsigma}$ in the following path

$$\hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}_c \rightsquigarrow \hat{\varsigma}_e \rightsquigarrow^* \hat{\varsigma}'_c \rightsquigarrow \hat{\varsigma}'_e \rightsquigarrow^* \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$$

where $\hat{\varsigma}'$ is an Exit-Esc over $[\![(k\ e)^\gamma]\!]$, $\hat{\varsigma}_e$ and $\hat{\varsigma}'_e$ are entries over $def_\lambda(k)$, $\hat{\varsigma}_c$ and $\hat{\varsigma}'_c$ are calls. The two entries have the form

$$\hat{\varsigma}_e = (def_\lambda(k), \hat{d}, \hat{c}, st, h)$$
$$\hat{\varsigma}'_e = (def_\lambda(k), \hat{d}', \hat{c}', st', h')$$

Both $\hat{c}$ and $\hat{c}'$ can flow to k and we can call either at $\hat{\varsigma}'$. If we choose to restore $\hat{c}$ and st then $\hat{\varsigma}$ is in the same procedure as $\hat{\varsigma}_c$. If we restore $\hat{c}'$ and st', $\hat{\varsigma}$ is in the same procedure as $\hat{\varsigma}'_c$. However, it is possible that $\hat{c} = \hat{c}'$ and $st = st'$, in which case $\hat{\varsigma}$ belongs to two procedures. Unique decomposition no longer holds.

For this reason, we now define a set of corresponding entries for each state, instead of a single entry [21].

Definition 5 (Corresponding Entries)**.**
Let $p \equiv \hat{\varsigma}_e \rightsquigarrow^* \hat{\varsigma}$ where $\hat{\varsigma}_e$ is an entry. We define $CE_p(\hat{\varsigma})$ to be the smallest set such that:

- *if $\hat{\varsigma}$ is an entry, $CE_p(\hat{\varsigma}) = \{\hat{\varsigma}\}$*
- *if $p \equiv \hat{\varsigma}_e \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}$, $\hat{\varsigma}$ is an Exit-Esc over $[\![(k\ e)^\gamma]\!]$, $\hat{\varsigma}_1$ is an entry over $def_\lambda(k)$, then $\hat{\varsigma}_1 \in CE_p(\hat{\varsigma})$.*
- *if $p \equiv \hat{\varsigma}_e \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}$, $\hat{\varsigma}$ is neither an entry nor an Exit-Esc, $\hat{\varsigma}_1$ is neither an Exit-Ret nor an Exit-Esc, then $CE_p(\hat{\varsigma}) = CE_p(\hat{\varsigma}_1)$.*
- *if $p \equiv \hat{\varsigma}_e \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^* \hat{\varsigma}_3 \rightsquigarrow^+ \hat{\varsigma}_4 \rightsquigarrow \hat{\varsigma}$, $\hat{\varsigma}$ is a $\widehat{CApply}$ of the form $(\hat{c}, \hat{d}, st, h)$, $\hat{\varsigma}_4$ is an Exit-Esc, $\hat{\varsigma}_3 \in CE_p(\hat{\varsigma}_4)$ and has the form $(ulam, \hat{d}', \hat{c}, st, h')$, $\hat{\varsigma}_2 \in CE^*_p(\hat{\varsigma}_3)$, $\hat{\varsigma}_1$ is a Call, then $CE_p(\hat{\varsigma}_1) \subseteq CE_p(\hat{\varsigma})$.*
- *if $p \equiv \hat{\varsigma}_e \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^+ \hat{\varsigma}_3 \rightsquigarrow \hat{\varsigma}$, $\hat{\varsigma}$ is a $\widehat{CApply}$, $\hat{\varsigma}_3$ is an Exit-Ret, $\hat{\varsigma}_2 \in CE^*_p(\hat{\varsigma}_3)$, $\hat{\varsigma}_1$ is a Call, then $CE_p(\hat{\varsigma}_1) \subseteq CE_p(\hat{\varsigma})$.*

For each state $\hat{\varsigma}$, we also define $CE^*_p(\hat{\varsigma})$ to be the set of entries that can reach an entry in $CE_p(\hat{\varsigma})$ through tail calls.

Definition 6. *Let $p \equiv \hat{\varsigma}_e \rightsquigarrow^* \hat{\varsigma}$ where $\hat{\varsigma}_e$ is an entry. We define $CE^*_p(\hat{\varsigma})$ to be the smallest set such that:*

- *$CE_p(\hat{\varsigma}) \subseteq CE^*_p(\hat{\varsigma})$*
- *if $p \equiv \hat{\varsigma}_e \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^* \hat{\varsigma}$, $\hat{\varsigma}_2 \in CE_p(\hat{\varsigma})$, $\hat{\varsigma}_1$ is a Tail Call, then $CE^*_p(\hat{\varsigma}_1) \subseteq CE^*_p(\hat{\varsigma})$.*

Note that if $\hat{\varsigma}$ is an Exit-Esc over $[\![(k\ e)^\gamma]\!]$, a procedure that contains $\hat{\varsigma}$ has an entry $\hat{\varsigma}'$ over $iu_\lambda([\![(k\ e)^\gamma]\!])$. Thus, $\hat{\varsigma}'$ is not in $CE_p(\hat{\varsigma})$ because $iu_\lambda([\![(k\ e)^\gamma]\!]) \neq def_\lambda(k)$. For all other states, $CE_p(\hat{\varsigma})$ is the set of entries of procedures that contain $\hat{\varsigma}$. The following lemma relates the stack of a state with the stacks of its corresponding entries.

Lemma 7.
Let $p \equiv \hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}$ where $\hat{\varsigma} \equiv (\dots, st, h)$.

1. *If $\hat{\varsigma}$ is a final state then $CE_p(\hat{\varsigma}) = \emptyset$.*
2. *If $\hat{\varsigma}$ is an entry then $CE_p(\hat{\varsigma}) \neq \emptyset$. (Thus, $CE^*_p(\hat{\varsigma}) \neq \emptyset$.)*
   Let $\hat{\varsigma}_e \in CE^*_p(\hat{\varsigma})$, of the form $(ulam, \hat{d}, \hat{c}, st_e, h_e)$. Then, $st = st_e$ and the continuation argument of $\hat{\varsigma}$ is $\hat{c}$.
3. *If $\hat{\varsigma}$ is an Exit-Esc then its stack is not empty and $CE_p(\hat{\varsigma}) \neq \emptyset$. (We do not assert anything about the stack change between a state in $CE^*_p(\hat{\varsigma})$ and $\hat{\varsigma}$, it can be arbitrary.)*
4. *If $\hat{\varsigma}$ is none of the above then $CE_p(\hat{\varsigma}) \neq \emptyset$.*
   Let $\hat{\varsigma}_e = ([\![(\lambda_l\,(u\,k)\ call)]\!], \hat{d}, \hat{c}, st_e, h_e)$.
   If $\hat{\varsigma}_e \in (CE^*_p(\hat{\varsigma}) \setminus CE_p(\hat{\varsigma}))$ then
   - *there is a frame tf such that $st \equiv tf :: st_e$.*
   - *there is a variable k' such that $tf(k') = \hat{c}$.*

   If $\hat{\varsigma}_e \in CE_p(\hat{\varsigma})$ then there is a frame tf such that $st \equiv tf :: st_e$, $\mathrm{dom}(tf) \subseteq LV(l)$, $tf(u) \sqsubseteq \hat{d}$, $tf(k) = \hat{c}$.

The proof of lemma 7 proceeds by induction on the length of the path p. We now state the soundness theorem. Its proof and the proof of lemma 7 can be found in the appendix.

Theorem 8 (Soundness)**.**
If $p \equiv \hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}$ then, after summarization:

- *If $\hat{\varsigma}$ is a final state then $|\hat{\varsigma}|_{al} \in Final$*
- *If $\hat{\varsigma}$ is not final and $\hat{\varsigma}' \in CE_p(\hat{\varsigma})$ then $(|\hat{\varsigma}'|_{al}, |\hat{\varsigma}|_{al}) \in Seen$*
- *If $\hat{\varsigma}$ is an Exit-Ret or Exit-Esc and $\hat{\varsigma}' \in CE^*_p(\hat{\varsigma})$ then $(|\hat{\varsigma}'|_{al}, |\hat{\varsigma}|_{al}) \in Seen$*

CFA2 without first-class control is complete, which means that there is no loss in precision when going from abstract to local states [22]. The algorithm of Fig. 8 is not complete; it may compute flows that never happen in the abstract semantics.

```
(define esc (λ(f cc) (f (λ(x k) (cc x)) cc)))

(esc (λ1(v1 k1) (v1 "foo" k1))
     (λ(a) (halt a)))

(esc (λ2(v2 k2) (k2 "bar"))
     (λ(b) (halt b)))
```

In this program, `esc` is the CPS translation of `call/cc`. The two user functions λ_1 and λ_2 expect a reified continuation as their first argument; λ_1 uses that continuation and λ_2 does not. The abstract semantics finds that {`"foo"`} flows to `a` and {`"bar"`} flows to `b`.

However, the workset algorithm thinks that that {`"foo"`,`"bar"`} flows to `b`. At the second call to `esc`, it connects the entry to the Exit-Esc state over [[`(cc x)`]] at line 11, which is a spurious flow.

5.5 Various approaches to downward continuations

In RCPS, the general form of a user lambda that binds a heap continuation variable is

```
(λ1(u k) (...(λ2(u2 k2) (...(k e)^γ ...)) ...))
```

where λ_1 contains a user lambda λ_2, which in turn contains a heap reference to `k` in operator position.

During execution, if a closure over λ_2 escapes upward, merging of continuations at $[\![(k\ e)^\gamma]\!]$ is unavoidable. However, when λ_2 is not passed upward, the abstract semantics still merges at $[\![(k\ e)^\gamma]\!]$. A natural question to ask is how precise can CFA2 be for downward continuations, either exception handlers or continuations captured by `call/cc` that never escape. In both cases, we can avoid merging.

In section 2.4, we saw how the JavaScript implementation of CFA2 handles exception throws precisely. Another way to achieve this is by uniformly passing two continuations to each user function, the current continuation and an exception handler [5]. Consider a user lambda $[\![(\lambda(\texttt{u k1 k2})\ (\ldots(\texttt{k2 e})^{\gamma}\ \ldots))]\!]$ where $S_?(\gamma, \texttt{k2})$ holds. Every Exit-Ret over $[\![(\texttt{k2 e})^{\gamma}]\!]$ is an exception throw. The handler continuation lives somewhere on the stack. To find it, we propagate transitive summaries for calls, like we do for tail calls. When the algorithm finds an edge $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ where $\tilde{\varsigma}_2$ is an Exit-Ret over $[\![(\texttt{k2 e})^{\gamma}]\!]$, it searches in *Callers* for a triple $(\tilde{\varsigma}_3, \tilde{\varsigma}_4, \tilde{\varsigma}_1)$. If the second continuation argument of $\tilde{\varsigma}_4$ is a lambda, we have found a handler. If not, we propagate a summary $(\tilde{\varsigma}_3, \tilde{\varsigma}_2)$, which has the effect of looking for a handler deeper in the stack. Note that the algorithm must keep these new summaries separate from the other summaries, to not confuse exceptional with ordinary control flow.

For continuations captured by `call/cc` that are only used downward, we can avoid merging by combining flow analysis and escape analysis. Consider the lambda at the beginning of this subsection. During flow analysis, we track if any closure over λ_2 escapes upward. We do that by checking for summaries $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$, where $\tilde{\varsigma}_1$ is an entry over λ_1. If λ_2 is contained in a binding reachable from $\tilde{\varsigma}_2$ (*cf.* [12, sec. 4.4.2]), then λ_2 is passed upward and we use the heap to look up `k` at $[\![(\texttt{k e})^{\gamma}]\!]$. Otherwise, we can assume that λ_2 does not escape. Hence, when we see an edge $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ where $\tilde{\varsigma}_1$ is an entry over λ_2 and $\tilde{\varsigma}_2$ is an Exit-Esc over $[\![(\texttt{k e})^{\gamma}]\!]$, we treat it as an exception throw. We use the new transitive summaries to search deeper in the stack for a live activation of λ_1, which tells us what flows to `k`.

6. Related work

The CFA2 workset algorithm is influenced by the functional approach of Sharir and Pnueli [16] and the tabulation algorithm of Reps *et al.* [14]. CFA2 extends these algorithms to first-class functions, introduces the stack/heap split and applies to control constructs that break call/return nesting. Traditional summary edges describe intraprocedural entry-to-exit flows. We have created several kinds of cross-procedure summaries for the various control patterns. Summaries for tail calls describe flows that do not grow the stack. Summaries for exceptions describe flows that grow the stack; the source of the summary may be deeper in the stack than the target. Finally, summaries for first-class control describe flows with arbitrary stack-change. The four different kinds of summaries can be conceptually unified because they serve a common purpose: *they connect a continuation passed to a user function with the state that calls it.*

Earl *et al.* proposed a pushdown higher-order flow analysis that does not use frames [8]. Instead, it allocates all bindings in the heap with context, in the style of *k*-CFA. For $k = 0$, their analysis runs in time $O(n^6)$, where n is the size of the program. Like all pushdown-reachability algorithms, Earl *et al.*'s analysis records pairs of states $(\varsigma_1, \varsigma_2)$ where ς_2 is same-context reachable from ς_1. However, their algorithm does not classify states as entries, exits, calls, *etc*. This has two drawbacks compared to the tabulation algorithm. First, they do not distinguish between path and summary edges. Thus, they have to search the whole set of edges when they look for return points, even though only summaries can contribute to the search. More importantly, path edges are only a small subset of the set S of all edges between same-context reachable states. By not classifying states, their algorithm maintains the whole set S, not just the path edges. In other words, it records edges whose source is not an entry. In the graph of Fig. 1, some of these edges are $\langle 2, 3\rangle$, $\langle 2, 6\rangle$, $\langle 5, 7\rangle$. Such edges slow down the analysis and do not contribute to call/return matching, because they cannot evolve into summary edges. In CFA2, it is possible to disable the use of frames by classifying each reference as a heap reference. The resulting analysis has similar precision to Earl *et al.*'s analysis for $k = 0$. We conjecture that this variant is not a viable alternative in practice, because of the significant loss in precision.

While there is extensive literature on finite-state higher-order flow analysis, little progress has been made in taming the power of `call/cc` and general continuations. Might and Shivers's ΔCFA [12, 13] introduced a notion of "frame strings" to track stack motion; these strings provide a notational vocabulary for describing and distinguishing various sorts of control transfer: recursive call, tail call, return, primitive application, as well as the more exotic control acts that are constructed with first-class control operators. However, the expressiveness of this device is brought low by its eventual regular-expression-based abstraction. Once abstracted, it loses much of its ability to reason about unusual patterns of control flow. We suspect that the infinite-state analytic framework provided by CFA2 could be the missing piece that would enable a ΔCFA-based analysis to be computed without requiring precision-destroying abstractions.

Shivers and Might have also shown how functional coroutines can be constructed with continuations, and then exploited to fuse pipelines of online transducers together into efficient, optimized code [18]. Being able to apply the power of pushdown models such as CFA2 to the transducer-fusion task raises interesting new possibilities. For example, suppose we had a coroutine generator with a recursive control structure—one that walks a binary tree producing the elements at the leaves. We wish to connect this tree-walking generator to a simple iterative coroutine that adds up all the items it receives. Is a pushdown flow analysis powerful enough to fuse the composition of these two coroutines into a single, recursive tree traversal, instead of an awkward, heavyweight implementation that ping-pongs back and forth between two independent stacks?

7. Conclusions

In this paper, we generalize the CFA2 flow analysis to first-class control. We propose an abstract semantics that allows stacks to be copied to the heap, and a summarization algorithm that handles the infinitely many heaps with a new kind of summary edges. With these additions, CFA2 becomes the first pushdown model that analyzes first-class control constructs. Moreover, CFA2 can now analyze the same language features as *k*-CFA, and do it more accurately. Thus, implementors of higher-order languages can use CFA2 as a drop-in replacement of *k*-CFA.

We also revisit the idea of path decomposition to accommodate states that belong to multiple procedures and prove our analysis sound. We show a program for which the abstract semantics gives a different result from the local semantics and conclude that our new summarization algorithm is not complete. We are not certain that first-class control unavoidably leads to incompleteness; we plan to investigate if changes to the algorithm can make it complete. However, it is possible that the abstract semantics describes a machine strictly more expressive than pushdown systems, and that reachability for this machine is not decidable.

Acknowledgments

This research was supported by a grant from the Mozilla Foundation.

References

[1] Doctor JS: a set of static analysis tools for JavaScript. https://github.com/mozilla/doctorjs.

[2] Generators in JavaScript. https://developer.mozilla.org/en/JavaScript/New_in_JavaScript/1.7.

[3] Generators in Python. http://www.python.org/dev/peps/pep-0255.

[4] The Programming Language Lua. http://www.lua.org.

[5] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[6] Ole-Johan Dahl and Kristen Nygaard. SIMULA - an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[7] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical Report 89/12, University of Copenhagen, 1989.

[8] Christopher Earl, Matthew Might, and David Van Horn. Pushdown Control-Flow Analysis of Higher-Order Programs. In *Workshop on Scheme and Functional Programming*, 2010.

[9] Matthias Felleisen. The Theory and Practice of First-Class Prompts. In *Principles of Programming Languages*, pages 180–190, 1988.

[10] David Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Yale University, 1988.

[11] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An Optimizing Compiler for Scheme. In *Compiler Construction*, pages 219–233, 1986.

[12] Matthew Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, 2007.

[13] Matthew Might and Olin Shivers. Analyzing the environment structure of higher-order languages using frame strings. *Theoretical Computer Science*, 375(1–3):137–168, May 2007.

[14] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Principles of Programming Languages*, pages 49–61, 1995.

[15] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *International Conference on Functional Programming*, pages 317–328, 2009.

[16] Micha Sharir and Amir Pnueli. Two Approaches to Interprocedural Data Flow Analysis.

[17] Olin Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Workshop on Continuations*, pages 1–15, 1997.

[18] Olin Shivers and Matthew Might. Continuations and transducer composition. In *Programming Language Design and Implementation*, pages 295–307, 2006.

[19] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews, editors. *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010.

[20] Guy L. Steele. Rabbit: A Compiler for Scheme. Master's thesis, MIT, 1978.

[21] Dimitrios Vardoulakis and Olin Shivers. CFA2: a Context-Free Approach to Control-Flow Analysis. In *European Symposium on Programming*, pages 570–589, 2010.

[22] Dimitrios Vardoulakis and Olin Shivers. CFA2: a Context-Free Approach to Control-Flow Analysis. *Logical Methods in Computer Science*, 7(2:3):1–39, May 2011.

[23] Dimitrios Vardoulakis and Olin Shivers. Ordering Multiple Continuations on the Stack. In *Workshop on Partial Evaluation and Program Manipulation*, pages 13–22, 2011.

[24] Mitchell Wand. Continuation-based multiprocessing. In *LISP and Functional Programming*, pages 19–28, 1980.

A.

We assume that CFA2 works on an alphatized program, *i.e.*, a program where all variables have distinct names. Thus, if an alphatized program contains a term $(\lambda_\psi(v_1 \ldots v_n)\, call)$, we know that no other lambda in that program binds variables with names $v_1, \ldots, v_n$. (During execution of CFA2, we do not rename any variables.) The following lemma is a simple consequence of alphatization.

Lemma 9. *A concrete state ς has the form $(\ldots, ve, t)$.*

1. *For any closure $(lam, \beta) \in \text{range}(ve)$, it holds that $\text{dom}(\beta) \cap BV(lam) = \emptyset$.*
2. *If ς is an Eval with call site call and environment β, then $\text{dom}(\beta) \cap BV(call) = \emptyset$.*
3. *If ς is an Apply, for any closure (lam, β) in operator or argument position, then $\text{dom}(\beta) \cap BV(lam) = \emptyset$.*

Proof. We show that the lemma holds for the initial state $\mathcal{I}(pr)$. Then, for each transition $\varsigma \to \varsigma'$, we assume that ς satisfies the lemma and show that ς' also satisfies it.

- $\mathcal{I}(pr)$ is a *UApply* of the form $((pr, \emptyset), (lam, \emptyset), halt, \emptyset, \langle\rangle)$. Since ve is empty, (1) trivially holds. Also, both closures have an empty environment so (3) holds.
- The [UEA] transition is:
  $([\![(f\ e\ q)^l]\!], \beta, ve, t) \to (proc, d, c, ve, l :: t)$
  $proc = \mathcal{A}(f, \beta, ve)$
  $d = \mathcal{A}(e, \beta, ve)$
  $c = \mathcal{A}(q, \beta, ve)$

  The ve doesn't change in the transition, so (1) holds for ς'.
  The operator is a closure of the form (lam, β'). We must show that $\text{dom}(\beta') \cap BV(lam) = \emptyset$. If $Lam_?(f)$, then $lam = f$ and $\beta' = \beta$. Also, we know
  $\text{dom}(\beta) \cap BV([\![(f\ e\ q)^l]\!]) = \emptyset$
  $\Rightarrow\ \text{dom}(\beta) \cap (BV(f) \cup BV(e) \cup BV(q)) = \emptyset$
  $\Rightarrow\ \text{dom}(\beta) \cap BV(f) = \emptyset$.
  If $Var_?(f)$, then $(lam, \beta') \in \text{range}(ve)$, so we get the desired result because ve satisfies (1).
  Similarly for d and c.
- The [UAE] transition is:
  $(proc, d, c, ve, t) \to (call, \beta', ve', t)$
  $proc \equiv \langle[\![(\lambda_l(u\,k)\ call)]\!], \beta\rangle$
  $\beta' = \beta[u \mapsto t][k \mapsto t]$
  $ve' = ve[(u,t) \mapsto d][(k,t) \mapsto c]$

  To show (1) for ve', it suffices to show that d and c don't violate the property. The user argument d is of the form (lam_1, β_1). Since ς satisfies (3), we know $\text{dom}(\beta_1) \cap BV(lam_1) = \emptyset$, which is the desired result. Similarly for c.
  Also, we must show that ς' satisfies (2). We know $\{u, k\} \cap BV(call) = \emptyset$ because the program is alphatized. Also, from property (3) for ς we know $\text{dom}(\beta) \cap BV([\![(\lambda_l(u\,k)\ call)]\!]) = \emptyset$, which implies $\text{dom}(\beta) \cap BV(call) = \emptyset$. We must show
  $\text{dom}(\beta') \cap BV(call) = \emptyset$
  $\Leftrightarrow\ (\text{dom}(\beta) \cup \{u,k\}) \cap BV(call) = \emptyset$
  $\Leftrightarrow\ (\text{dom}(\beta) \cap BV(call)) \cup (\{u,k\} \cap BV(call)) = \emptyset$
  $\Leftrightarrow\ \emptyset \cup \emptyset = \emptyset$.
- Similarly for the other two transitions. □

Theorem 10 (Simulation). *If $\varsigma \to \varsigma'$ and $|\varsigma|_{ca} \sqsubseteq \hat{\varsigma}$, then there exists $\hat{\varsigma}'$ such that $\hat{\varsigma} \leadsto \hat{\varsigma}'$ and $|\varsigma'|_{ca} \sqsubseteq \hat{\varsigma}'$.*

Proof. By cases on the concrete transition.

Rule [UAE]:
$(proc, d, c, ve, t) \to (call, \beta', ve', t)$
$proc \equiv \langle[\![(\lambda_l(u\,k)\ call)]\!], \beta\rangle$
$\beta' = \beta[u \mapsto t][k \mapsto t]$
$ve' = ve[(u,t) \mapsto d][(k,t) \mapsto c]$

Let $ts = \begin{cases} \langle\rangle & c = halt \\ toStack(LV(\mathcal{L}(lam)), \beta_1, ve) & c = (lam, \beta_1) \end{cases}$

Since $|\varsigma|_{ca} \sqsubseteq \hat{\varsigma}$, $\hat{\varsigma}$ is of the form $([\![(\lambda_l(u\,k)\ call)]\!], \hat{d}, \hat{c}, st, h)$, where

$$|d|_{ca} \sqsubseteq \hat{d}, \quad |c|_{ca} = \hat{c}, \quad ts \sqsubseteq st, \quad |ve|_{ca} \sqsubseteq h \qquad (1)$$

The abstract transition is
$([\![(\lambda_l(u\,k)\ call)]\!], \hat{d}, \hat{c}, st, h) \leadsto (call, st', h')$
$st' = push([u \mapsto \hat{d}][k \mapsto \hat{c}], st)$
$h'(v) = \begin{cases} h(u) \cup \hat{d} & (v = u) \wedge H_?(u) \\ h(k) \cup \{(\hat{c}, st)\} & (v = k) \wedge H_?(k) \\ h(v) & o/w \end{cases}$

Let ts' be the stack of $|\varsigma'|_{ca}$. The innermost user lambda that contains $call$ is λ_l, therefore $ts' = toStack(LV(l), \beta', ve')$. We must show that $|\varsigma'|_{ca} \sqsubseteq \hat{\varsigma}'$, *i.e.*,

$$ts' \sqsubseteq st' \qquad (2)$$

and

$$|ve'|_{ca} \sqsubseteq h' \qquad (3)$$

We assume that $c = (lam, \beta_1)$. (Showing (2) and (3) for $c = halt$ is similar.) We start with (3). We know that $|ve'|_{ca}$ is the same as $|ve|_{ca}$, except potential bindings for u and k.
If $H_?(u)$ holds, we must show
$|ve'|_{ca}(u) \sqsubseteq h'(u)$
$\Leftarrow\ |ve|_{ca}(u) \cup |d|_{ca} \sqsubseteq h(u) \cup \hat{d}$
$\Leftarrow\ |ve|_{ca} \sqsubseteq h\ \wedge\ |d|_{ca} \sqsubseteq \hat{d}$
$\Leftarrow$ (1)
If $H_?(k)$, then we must show $|ve'|_{ca}(k) \sqsubseteq h'(k)$. We know that $|ve'|_{ca}(k) = |ve|_{ca}(k) \cup \langle lam, toStack(LV(\mathcal{L}(lam)), \beta_1, ve')\rangle$ and $h'(k) = h(k) \cup \langle lam, st\rangle$.
By (1), it is enough to show $toStack(LV(\mathcal{L}(lam)), \beta_1, ve') \sqsubseteq st$. Since $ts \sqsubseteq st$, it suffices to show

$$ts = toStack(LV(\mathcal{L}(lam)), \beta_1, ve') \qquad (4)$$

By the temporal consistency of states (*cf.* [12] def. 4.4.5), (4) holds because the two bindings of ve' born at time t are younger than all bindings in β_1.

We proceed to show (2). We know that β' contains bindings for u and k, and by lemma 9 it doesn't bind any variables in $BV(call)$. Since $LV(l) \setminus \{u,k\} = BV(call)$, β' doesn't bind any variables in $LV(l) \setminus \{u,k\}$. Thus, the top frame of ts' is $[u \mapsto |d|_{ca}][k \mapsto |c|_{ca}]$. The top frame of st' is $[u \mapsto \hat{d}][k \mapsto \hat{c}]$, therefore the frames are in $\sqsubseteq$. To complete the proof of (2), we must show that
$pop(ts') \sqsubseteq pop(st')$
$\Leftarrow\ pop(ts') \sqsubseteq st$
$\stackrel{(1)}{\Leftarrow}\ pop(ts') = ts$
But $pop(ts') = toStack(LV(\mathcal{L}(lam)), \beta_1, ve')$, so by (4) we get the desired result.

Rule [CEA]:
$([\![(q\ e)^\gamma]\!], \beta, ve, t) \to (proc, d, ve, \gamma :: t)$
$proc = \mathcal{A}(q, \beta, ve)$
$d = \mathcal{A}(e, \beta, ve)$

Let $ts = toStack(LV(\gamma), \beta, ve)$. Since $|\varsigma|_{ca} \sqsubseteq \hat{\varsigma}$, $\hat{\varsigma}$ is of the form $([\![(q\ e)^{\gamma}]\!], st, h)$, where

$$|ve|_{ca} \sqsubseteq h, \quad ts \sqsubseteq st \tag{5}$$

The abstract transition is
$([\![(q\ e)^{\gamma}]\!], st, h) \rightsquigarrow (\hat{c}, \hat{d}, st', h)$
$\hat{d} = \hat{\mathcal{A}}_u(e, \gamma, st, h)$

$$(\hat{c}, st') \in \begin{cases} \{(q, st)\} & Lam_?(q) \\ \{(st(q), pop(st))\} & S_?(\gamma, q) \\ h(q) & H_?(\gamma, q) \end{cases}$$

Let ts' be the stack of $|\varsigma'|_{ca}$. We must show $|\varsigma'|_{ca} \sqsubseteq \hat{\varsigma}'$, *i.e.*,

$$|proc|_{ca} = \hat{c} \tag{6}$$

$$|d|_{ca} \sqsubseteq \hat{d} \tag{7}$$

$$ts' \sqsubseteq st' \tag{8}$$

Showing (7) is simple, by cases on e.
We will show (6) and (8) simultaneously, by cases on q:

- $Lam_?(q)$
  Then, $proc = (q, \beta)$ and $\hat{c} = q$, which imply (6).
  Also, $st' = st$, so (8) follows from $ts' \sqsubseteq st$
  $\Leftarrow\ toStack(LV(\mathcal{L}(q)), \beta, ve) \sqsubseteq st$
  $\Leftarrow\ toStack(LV(\gamma), \beta, ve) \sqsubseteq st$
  $\Leftarrow$ (5)
- $S_?(\gamma, q)$ and $proc = ve(q, \beta(q)) = (lam, \beta_1)$
  For (6), it suffices to show $\hat{c} = lam$
  $\Leftarrow\ st(q) = lam$
  $\stackrel{(5)}{\Leftarrow}\ ts(q) = lam$
  $\Leftarrow\ q \in LV(\gamma)$
  which holds because $S_?(\gamma, q)$.
  For (8), it suffices to show
  $toStack(LV(\mathcal{L}(lam)), \beta_1, ve) \sqsubseteq pop(st)$
  $\Leftarrow\ pop(ts) \sqsubseteq pop(st)$
  $\Leftarrow$ (5)
- $S_?(\gamma, q)$ and $proc = ve(q, \beta(q)) = halt$
  Similar to the previous case.
- $H_?(\gamma, q)$ and $proc = ve(q, \beta(q)) = (lam, \beta_1)$
  In this case, $|proc|_{ca} = lam$ and
  $(lam, toStack(LV(\mathcal{L}(lam)), \beta_1, ve)) \in |ve|_{ca}(q)$.
  By (5), there exists a pair $(lam, st') \in h(q)$ such that
  $toStack(LV(\mathcal{L}(lam)), \beta_1, ve) \sqsubseteq st'$.
  By picking this pair for $\hat{\varsigma}'$, we get (6) and (8) because
  $ts' = toStack(LV(\mathcal{L}(lam)), \beta_1, ve)$.
- $H_?(\gamma, q)$ and $proc = ve(q, \beta(q)) = halt$
  Similar to the previous case. □

Lemma 11 (Same-level reachability)**.**
Let $p \equiv \hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}$ *where* $\hat{\varsigma} \equiv (\ldots, st, h)$.

1. *If* $\hat{\varsigma}$ *is a final state then* $CE_p(\hat{\varsigma}) = \emptyset$.
2. *If* $\hat{\varsigma}$ *is an entry then* $CE_p(\hat{\varsigma}) \neq \emptyset$. *(Thus,* $CE^*_p(\hat{\varsigma}) \neq \emptyset$.*)*
   Let $\hat{\varsigma}_e \in CE^*_p(\hat{\varsigma})$*, of the form* $(ulam, \hat{d}, \hat{c}, st_e, h_e)$*. Then,* $st = st_e$ *and the continuation argument of* $\hat{\varsigma}$ *is* $\hat{c}$.
3. *If* $\hat{\varsigma}$ *is an Exit-Esc then its stack is not empty and* $CE_p(\hat{\varsigma}) \neq \emptyset$. *(We do not assert anything about the stack change between a state in* $CE^*_p(\hat{\varsigma})$ *and* $\hat{\varsigma}$*, it can be arbitrary.)*
4. *If* $\hat{\varsigma}$ *is none of the above then* $CE_p(\hat{\varsigma}) \neq \emptyset$.
   Let $\hat{\varsigma}_e = ([\![(\lambda_l\,(u\,k)\ call)]\!], \hat{d}, \hat{c}, st_e, h_e)$.
   If $\hat{\varsigma}_e \in (CE^*_p(\hat{\varsigma}) \setminus CE_p(\hat{\varsigma}))$ *then*
   - *there is a frame* tf *such that* $st \equiv tf :: st_e$.
   - *there is a variable* k' *such that* $tf(k') = \hat{c}$.

   If $\hat{\varsigma}_e \in CE_p(\hat{\varsigma})$ *then*
   - *there is a frame* tf *such that* $st \equiv tf :: st_e, \mathrm{dom}(tf) \subseteq LV(l), tf(u) \sqsubseteq \hat{d}, tf(k) = \hat{c}$.
   - *if* $\hat{\varsigma}$ *is an* $\widehat{Eval}$ *over a call site labeled* ψ *then* $\psi \in LL(l)$.
   - *if* $\hat{\varsigma}$ *is a* $\widehat{CApply}$ *over a lambda labeled* γ *then* $\gamma \in LL(l)$.

Proof. By induction on the length of p.
If the length is 0, then $\hat{\varsigma} = \hat{\mathcal{I}}(pr)$. By definitions 5 and 6, $CE_p(\hat{\varsigma}) = CE^*_p(\hat{\varsigma}) = \{\hat{\mathcal{I}}(pr)\}$ and the lemma trivially holds.

If the length is greater than 0, p has the form $\hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$. We take cases on $\hat{\varsigma}$.

$\hat{\varsigma}$ is an entry.
By def. 5, $CE_p(\hat{\varsigma}) = \{\hat{\varsigma}\}$. If $\hat{\varsigma}'$ is a call, then by def. 6, $CE^*_p(\hat{\varsigma}) = \{\hat{\varsigma}\}$. The lemma trivially holds in this case.

If $\hat{\varsigma}'$ is a tail call, let a state $\hat{\varsigma}_e$ of the form $(ulam, \hat{d}, \hat{c}, st_e, h_e)$ be in $CE^*_p(\hat{\varsigma}')$. Thus, $\hat{\varsigma}_e \in CE^*_p(\hat{\varsigma})$ by def. 6. We must show that $st = st_e$ and the continuation argument of $\hat{\varsigma}$ is $\hat{c}$. By *IH*, the stack st' of $\hat{\varsigma}'$ has the form $tf :: st_e$ and there is a variable k such that $tf(k) = \hat{c}$. By rule $[\widehat{\mathrm{UEA}}]$, $st = st_e$ and the continuation argument of $\hat{\varsigma}$ is $\hat{c}$.

$\hat{\varsigma}$ is an Exit-Esc.
Let $[\![(k\ e)^{\gamma}]\!]$ be the call site in $\hat{\varsigma}$. The set $h(k)$ contains pairs of the form $(\hat{c}', st')$. Each such pair can only be put in h when transitioning from a $\widehat{UApply}$ over $def_{\lambda}(k)$ to an $\widehat{Eval}$. Each such $\widehat{UApply}$ is in $CE_p(\hat{\varsigma})$.

We must show that the stack of $\hat{\varsigma}$ is not empty. The predecessor $\hat{\varsigma}'$ of $\hat{\varsigma}$ is an $\widehat{Apply}$. If $\hat{\varsigma}'$ is a $\widehat{UApply}$ then by rule $[\widehat{\mathrm{UAE}}]$ the stack of $\hat{\varsigma}$ has at least one frame. If $\hat{\varsigma}'$ a $\widehat{CApply}$ then by *IH* we get that $CE_p(\hat{\varsigma}') \neq \emptyset$ and that the stack of $\hat{\varsigma}'$ has one more frame than the stack of any state in $CE_p(\hat{\varsigma}')$. Thus, by rule $[\widehat{\mathrm{CAE}}]$, the stack of $\hat{\varsigma}$ is also non-empty.

$\hat{\varsigma}$ is a $\widehat{CApply}$ and $\hat{\varsigma}'$ is an Exit-Esc.
The two states have the form:
$\hat{\varsigma} = (\hat{c}, \hat{d}, st, h)$
$\hat{\varsigma}' = ([\![(k\ e)^{\gamma}]\!], st', h)$
By *IH*, $CE^*_p(\hat{\varsigma}') \neq \emptyset$. All entries in $CE^*_p(\hat{\varsigma}')$ are over $def_{\lambda}(k)$. Since $\hat{\varsigma}$ is over $\hat{c}$ and has stack st, there is one or more entries in $CE^*_p(\hat{\varsigma}')$ whose stack is st and their continuation argument is $\hat{c}$.

Let S be the set of those entries. We first show that one of the two following statements holds.

- For each $\hat{\varsigma}_1$ in S, $\hat{\mathcal{I}}(pr) \in CE^*_p(\hat{\varsigma}_1)$.
- For each $\hat{\varsigma}_1$ in S, $\hat{\mathcal{I}}(pr) \notin CE^*_p(\hat{\varsigma}_1)$.

For the sake of contradiction, let $\hat{\varsigma}_1, \hat{\varsigma}_2 \in S$, such that $\hat{\mathcal{I}}(pr) \in CE^*_p(\hat{\varsigma}_1)$ and $\hat{\mathcal{I}}(pr) \notin CE^*_p(\hat{\varsigma}_2)$. Then, let $\hat{\varsigma}_3 \neq \hat{\mathcal{I}}(pr)$ be the earliest state in $CE^*_p(\hat{\varsigma}_2)$. Since it's the earliest, its predecessor $\hat{\varsigma}_4$ is a call. By *IH* for $\hat{\mathcal{I}}(pr) \rightsquigarrow^+ \hat{\varsigma}_2$, the stack of $\hat{\varsigma}_3$ is st and its continuation argument is $\hat{c}$. Then, since $\hat{\varsigma}_4$ is a call, $\hat{c}$ is the continuation lambda appearing at $\hat{\varsigma}_4$. Also, by *IH* for $\hat{\mathcal{I}}(pr) \rightsquigarrow^+ \hat{\varsigma}_1$, the continuation argument of $\hat{\varsigma}_1$ is $halt$. But then, $\hat{c}$ is simultaneously a lambda and $halt$, contradiction.

Now we prove the lemma considering only the two cases for S.

- For each $\hat{\varsigma}_1$ in S, $\hat{\mathcal{I}}(pr) \in CE^*_p(\hat{\varsigma}_1)$.
  In this case, $\hat{c} = halt$ and $st = \langle\rangle$. Thus, $\hat{\varsigma}$ is a final state. We must show that $CE^*_p(\hat{\varsigma}) = \emptyset$. By def. 5, if $CE^*_p(\hat{\varsigma})$ is not empty, then the path can be decomposed according to the fourth case:
  $p \equiv \hat{\mathcal{I}}(pr) \rightsquigarrow^+ \hat{\varsigma}_3 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$
  where $\hat{\varsigma}_2 \in CE^*_p(\hat{\varsigma}_1)$, $\hat{\varsigma}_3$ is a call, $CE_p(\hat{\varsigma}_3) \subseteq CE_p(\hat{\varsigma})$. But by *IH*, the continuation of $\hat{\varsigma}_2$ is $halt$, which is impossible because its predecessor is a call. Thus, $CE^*_p(\hat{\varsigma}) = \emptyset$.

- For each $\hat{\varsigma}_1$ in S, $\hat{\mathcal{I}}(pr) \notin CE^*_p(\hat{\varsigma}_1)$.
  Let $\hat{\varsigma}_2 \neq \hat{\mathcal{I}}(pr)$ be the earliest state in $CE^*_p(\hat{\varsigma}_1)$. Then, its predecessor $\hat{\varsigma}_3$ is a call. Thus, p has the form
  $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}_3 \leadsto \hat{\varsigma}_2 \leadsto^* \hat{\varsigma}_1 \leadsto^+ \hat{\varsigma}' \leadsto \hat{\varsigma}$
  By *IH* for $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}_1$, we get that the continuation argument of $\hat{\varsigma}_2$ is $\hat{c}$ and its stack is st. Then, by rule $[\widehat{\text{UEA}}]$, we get that $\hat{c}$ is the continuation lambda appearing at the call site of $\hat{\varsigma}_3$. Thus, $\hat{\varsigma}$ is not a final state, so we must show that $CE_p(\hat{\varsigma}) \neq \emptyset$.
  By the fourth item of def. 5, $CE_p(\hat{\varsigma}_3) \subseteq CE_p(\hat{\varsigma})$. But by *IH* for $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}_3$, we get that $CE_p(\hat{\varsigma}_3) \neq \emptyset$. Thus, $CE_p(\hat{\varsigma}) \neq \emptyset$.
  We now proceed to prove the remaining obligations for the states in $CE_p(\hat{\varsigma})$.
  Let $\hat{\varsigma}_e \in CE_p(\hat{\varsigma})$, of the form $([\![(\lambda_l (u\, k')\, call)]\!], \hat{d}_e, \hat{c}_e, st_e, h_e)$.
  $\hat{\varsigma}_3$ has the form $([\![(e_1\ e_2\ q)^{l'}]\!], st_3, h_3)$ where $q = \hat{c}$.
  By *IH* for $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}_3$, we get that $st_3 = tf :: st_e, \text{dom}(tf) \subseteq LV(l), tf(u) \sqsubseteq \hat{d}_e, tf(k') = \hat{c}_e, l' \in LL(l)$.
  $\hat{\varsigma}_2$ has the form $(ulam, \hat{d}_2, \hat{c}, st, h)$ where $st = tf' :: st_e$ and
  $$tf' = \begin{cases} tf & Lam_?(e_1) \vee H_?(l', e_1) \\ tf[e_1 \mapsto \{ulam\}] & S_?(l', e_1) \end{cases}$$
  We can see that the stack of $\hat{\varsigma}$ has the appropriate form: $tf'(k') = tf(k') = \hat{c}$ and $tf'(u) \sqsubseteq tf(u) \sqsubseteq \hat{d}_e$. Also, $\hat{c}$ is a lambda appearing at l', so $\mathcal{L}(\hat{c}) \in LL(l)$.
  For the case where $\hat{\varsigma}_e \in CE^*_p(\hat{\varsigma}_3) \setminus CE_p(\hat{\varsigma}_3)$, the proof is similar and simpler.

$\hat{\varsigma}$ is a $\widehat{CApply}$ and $\hat{\varsigma}'$ is an Exit-Ret.
The two states have the form:
$\hat{\varsigma} = (\hat{c}, \hat{d}, st, h)$
$\hat{\varsigma}' = ([\![(k\ e)^{\gamma}]\!], st', h)$
By *IH* for $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}'$, we get $CE_p(\hat{\varsigma}') \neq \emptyset, CE^*_p(\hat{\varsigma}') \neq \emptyset$.
Also, by *IH* and rule $[\widehat{\text{CEA}}]$, $st' = tf :: st, tf(k) = \hat{c}$.

- $\hat{\mathcal{I}}(pr) \in CE^*_p(\hat{\varsigma}')$
  By *IH* for $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}'$, we get $\hat{c} = halt, st = \langle\rangle$. Thus, $\hat{\varsigma}$ is a final state, so we must show $CE_p(\hat{\varsigma}) = \emptyset$. Assume that $CE_p(\hat{\varsigma}) \neq \emptyset$. This can only happen if the fifth item of def. 5 applies. In this case, p has the form
  $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}_1 \leadsto \hat{\varsigma}_2 \leadsto^+ \hat{\varsigma}' \leadsto \hat{\varsigma}$
  where $\hat{\varsigma}_1$ is a call and $\hat{\varsigma}_2 \in CE^*_p(\hat{\varsigma}')$. By *IH* for $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}'$, we get that the continuation argument of $\hat{\varsigma}_2$ is *halt* and by rule $[\widehat{\text{UEA}}]$ we get that it is the lambda passed in $\hat{\varsigma}_1$, which is a contradiction. Therefore, $CE_p(\hat{\varsigma}) = \emptyset$.
- $\hat{\mathcal{I}}(pr) \notin CE^*_p(\hat{\varsigma}')$
  Let $\hat{\varsigma}_2 \neq \hat{\mathcal{I}}(pr)$ be the earliest state in $CE^*_p(\hat{\varsigma}')$. The predecessor $\hat{\varsigma}_3$ of $\hat{\varsigma}_2$ is a call. By *IH* for $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}'$, we get that the continuation argument of $\hat{\varsigma}_2$ is $\hat{c}$ and its stack is st. By rule $[\widehat{\text{UEA}}]$, we get that $\hat{c}$ is the continuation lambda appearing at the call site of $\hat{\varsigma}_1$. Thus, $\hat{\varsigma}$ is not a final state, so we must show that $CE_p(\hat{\varsigma}) \neq \emptyset$. But by the fifth item of def. 5 we know that $CE_p(\hat{\varsigma}_1) \subseteq CE_p(\hat{\varsigma})$ and by *IH* we know that $CE^*_p(\hat{\varsigma}_1) \neq \emptyset$.
  To show that the stack of $\hat{\varsigma}$ has the desirable form, we work in the same way as in the case where $\hat{\varsigma}$ is a $\widehat{CApply}$ and $\hat{\varsigma}'$ is an Exit-Esc.

$\hat{\varsigma}$ is none of the above.
In this case, $\hat{\varsigma}$ is one of: $\widehat{UEval}$, inner $\widehat{CEval}$, Exit-Ret, $\widehat{CApply}$ whose predecessor is an inner $\widehat{CEval}$. The path can be decomposed as $\hat{\mathcal{I}}(pr) \leadsto^* \hat{\varsigma}' \leadsto \hat{\varsigma}$. By *IH*, $CE_p(\hat{\varsigma}') \neq \emptyset$, and by the third item of def. 5, $CE_p(\hat{\varsigma}) \neq \emptyset$. It's simple to show that the stack of $\hat{\varsigma}$ has the desirable properties by assuming that the stack of $\hat{\varsigma}'$ has them. □

Lemma 12 (Local simulation)**.**
If $\hat{\varsigma} \leadsto \hat{\varsigma}'$ and $succ(|\hat{\varsigma}|_{al}) \neq \emptyset$ then $|\hat{\varsigma}'|_{al} \in succ(|\hat{\varsigma}|_{al})$.

Theorem 13 (Soundness)**.**
If $p \equiv \hat{\mathcal{I}}(pr) \leadsto^* \hat{\varsigma}$ then, after summarization:

- *If $\hat{\varsigma}$ is a final state then $|\hat{\varsigma}|_{al} \in Final$*
- *If $\hat{\varsigma}$ is not final and $\hat{\varsigma}' \in CE_p(\hat{\varsigma})$ then $(|\hat{\varsigma}'|_{al}, |\hat{\varsigma}|_{al}) \in Seen$*
- *If $\hat{\varsigma}$ is an Exit-Ret or Exit-Esc and $\hat{\varsigma}' \in CE^*_p(\hat{\varsigma})$ then $(|\hat{\varsigma}'|_{al}, |\hat{\varsigma}|_{al}) \in Seen$*
- *If $\hat{\varsigma}$ is an Exit-Esc and $\hat{\varsigma}' \in CE^*_p(\hat{\varsigma})$ then $(|\hat{\varsigma}'|_{al}, |\hat{\varsigma}|_{al})$ is already in Summary when it is removed from W to be examined*

Proof. By induction on the length of p.
The basecase is simple.
If the length is greater than 0, p has the form $\hat{\mathcal{I}}(pr) \leadsto^* \hat{\varsigma}' \leadsto \hat{\varsigma}$. We take cases on $\hat{\varsigma}$.

$\hat{\varsigma}$ is an entry.
Then, $CE_p(\hat{\varsigma}) = \{\hat{\varsigma}\}$. Also, $\hat{\varsigma}'$ is a call or a tail call.
By lemma 11, $CE_p(\hat{\varsigma}') \neq \emptyset$. Let $\hat{\varsigma}_1 \in CE_p(\hat{\varsigma}')$. Then, p can be decomposed as $\hat{\mathcal{I}}(pr) \leadsto^* \hat{\varsigma}_1 \leadsto^+ \hat{\varsigma}' \leadsto \hat{\varsigma}$. By *IH*, $(|\hat{\varsigma}_1|_{al}, |\hat{\varsigma}'|_{al})$ was put in *Seen* at some point during the execution, so it was also put in W and examined. By lemma 12, $|\hat{\varsigma}|_{al} \in succ(|\hat{\varsigma}'|_{al})$ so in line 16 or 36 $(|\hat{\varsigma}|_{al}, |\hat{\varsigma}|_{al})$ will be propagated.

$\hat{\varsigma}$ is a $\widehat{CApply}$ and $\hat{\varsigma}'$ is an Exit-Esc.
The two states have the form:
$\hat{\varsigma} = (\hat{c}, \hat{d}, st, h)$
$\hat{\varsigma}' = ([\![(k\ e)^{\gamma}]\!], st', h)$
By lemma 11, $CE^*_p(\hat{\varsigma}') \neq \emptyset$. Since $(\hat{c}, st) \in h(k)$, there is a state $\hat{\varsigma}_1$ in $CE_p(\hat{\varsigma}')$ of the form:
$\hat{\varsigma}_1 = (def_\lambda(k), \hat{d}_1, \hat{c}, st, h_1)$
Also, by $H_?(\gamma, k)$ we get $def_\lambda(k) \neq pr$, which implies $\hat{\varsigma}_1 \neq \hat{\mathcal{I}}(pr)$. Thus, p can be written $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}_1 \leadsto^+ \hat{\varsigma}' \leadsto \hat{\varsigma}$. We take two cases.

- $\hat{\mathcal{I}}(pr) \in CE^*_p(\hat{\varsigma}_1)$
  In this case, $\hat{\mathcal{I}}(pr) \in CE^*_p(\hat{\varsigma}')$. By lemma 11 for $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}_1$, we get $\hat{c} = halt$ and $st = \langle\rangle$. Thus, $\hat{\varsigma}$ is a final state. By *IH*, $(|\hat{\mathcal{I}}(pr)|_{al}, |\hat{\varsigma}'|_{al})$ was put in *Summary* before it was put in *Seen*. Therefore, when it was examined at line 25, it was already in *Summary* and the test at line 26 was false.
  The test at line 30 was true, so `Final`$(|\hat{\varsigma}'|_{al})$ was called. By lemma 11 for $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}'$, we get that st' is not empty, so it has the form $tf :: st''$. Then, $|\hat{\varsigma}'|_{al}$ is
  $([\![(k\ e)^{\gamma}]\!], tf \upharpoonright UVar, h \upharpoonright UVar)$.
  At line 54, $(halt, \tilde{\mathcal{A}}_u(e, \gamma, tf \upharpoonright UVar, h \upharpoonright UVar), \emptyset, h \upharpoonright UVar)$ goes in *Final*. But this state is $|\hat{\varsigma}|_{al}$ because
  $\tilde{\mathcal{A}}_u(e, \gamma, tf \upharpoonright UVar, h \upharpoonright UVar)$ is equal to $\hat{\mathcal{A}}_u(e, \gamma, st', h)$.
- $\hat{\mathcal{I}}(pr) \notin CE^*_p(\hat{\varsigma}_1)$
  Let $\hat{\varsigma}_2 \neq \hat{\mathcal{I}}(pr)$ be the earliest state in $CE^*_p(\hat{\varsigma}_1)$. (Thus, $\hat{\varsigma}_2 \in CE^*_p(\hat{\varsigma}')$.) The predecessor $\hat{\varsigma}_3$ of $\hat{\varsigma}_2$ is a call. Thus,
  $p \equiv \hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}_3 \leadsto \hat{\varsigma}_2 \leadsto^* \hat{\varsigma}_1 \leadsto^+ \hat{\varsigma}' \leadsto \hat{\varsigma}$
  By lemma 11 for $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}_1$, we find that the continuation argument of $\hat{\varsigma}_2$ is $\hat{c}$ and its stack is st. By $[\widehat{\text{UEA}}]$, $\hat{c}$ is the continuation lambda passed at $\hat{\varsigma}_3$. Therefore, $\hat{\varsigma}$ is not a final state. By def. 5 we know that $CE_p(\hat{\varsigma}_3) \subseteq CE_p(\hat{\varsigma})$. For each $\hat{\varsigma}_4 \in CE_p(\hat{\varsigma}_3)$, we must show that $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}|_{al})$ was put in *Seen*. By *IH*, we know that $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}_3|_{al})$ and $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}'|_{al})$ were put in W and examined. We take cases on which edge was examined first.
  Assume that $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}_3|_{al})$ was examined first. By lemma 12, $|\hat{\varsigma}_2|_{al} \in succ(|\hat{\varsigma}_3|_{al})$, so in line 17 we put $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}_3|_{al}, |\hat{\varsigma}_2|_{al})$ in *Callers*. We later examine $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}'|_{al})$. By *IH*, $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}'|_{al})$

is in $Summary$ when it is examined, so the test at line 26 is false. Also, $\hat{\varsigma}_2 \neq \hat{\mathcal{I}}(pr)$ so the test at line 30 is false as well. Since $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}_3|_{al}, |\hat{\varsigma}_2|_{al})$ is in $Callers$, at line 32 we call `Update(`$|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}_3|_{al}, |\hat{\varsigma}_2|_{al}, |\hat{\varsigma}'|_{al}$`)`. We must show that the state $\tilde{\varsigma}$ constructed by `Update` is the same as $|\hat{\varsigma}|_{al}$.

By lemma 11 for $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}'$, we get that st' is not empty. It is easy to see that the user value passed at $\tilde{\varsigma}$, which is $\tilde{\mathcal{A}}_u(e, \gamma, |st'|_{al}, |h|_{al})$, is equal to $\hat{\mathcal{A}}_u(e, \gamma, st', h)$.

By lemma 11 for $\hat{\mathcal{I}}(pr) \leadsto^+ \hat{\varsigma}_3$, the stack of $\hat{\varsigma}_3$ is not empty. Thus, $\hat{\varsigma}_3$ has the form $(\llbracket (e_1\, e_2\, q)^l \rrbracket, tf :: st_3, h_3)$ where $q = \hat{c}$. Let $ulam$ be the function applied at $\hat{\varsigma}_2$. Then, by rule $[\widehat{\text{UEA}}]$, the stack st of $\hat{\varsigma}_2$ is

$$st = \begin{cases} tf :: st_3 & Lam_?(e_1) \vee H_?(l, e_1) \\ tf[e_1 \mapsto \{ulam\}] :: st_3 & S_?(l, e_1) \end{cases}$$

But then, $|st|_{al}$ is equal to the frame constructed at line 50. Therefore, $\tilde{\varsigma} = |\hat{\varsigma}|_{al}$.

Assume that $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}'|_{al})$ was examined first. In this case, when $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}_3|_{al})$ is examined, we call `Update` at line 18. The proof is similar.

$\hat{\varsigma}$ is an Exit-Esc.

Then, $\hat{\varsigma}'$ is an $\widehat{Apply}$. By lemma 11, $CE_p(\hat{\varsigma}') \neq \emptyset$. Let $\hat{\varsigma}_1 \in CE_p(\hat{\varsigma}')$. By *IH*, $(|\hat{\varsigma}_1|_{al}, |\hat{\varsigma}'|_{al})$ was examined. Also, by lemma 12, $|\hat{\varsigma}|_{al} \in succ(\hat{\varsigma}')$. Thus, in line 7 or 13, $(|\hat{\varsigma}_1|_{al}, |\hat{\varsigma}|_{al})$ was propagated but not put in $Summary$.

By lemma 11, $CE_p(\hat{\varsigma}) \neq \emptyset$. Let $\hat{\varsigma}_2 \in CE_p(\hat{\varsigma})$. By *IH*, $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}_2|_{al})$ was examined.

We proceed by cases on whether $(|\hat{\varsigma}_1|_{al}, |\hat{\varsigma}|_{al})$ or $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}_2|_{al})$ was examined first.

- $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}_2|_{al})$ was first
  When $(|\hat{\varsigma}_1|_{al}, |\hat{\varsigma}|_{al})$ is examined, the test at line 26 is true. Also, $|\hat{\varsigma}_2|_{al}$ is in $EntriesEsc$, it was put at line 10 when $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}_2|_{al})$ was examined. Thus, at line 29, $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}|_{al})$ is put in $Summary$ and $Seen$.
- $(|\hat{\varsigma}_1|_{al}, |\hat{\varsigma}|_{al})$ was first
  At line 27, $|\hat{\varsigma}|_{al}$ was put in $Escapes$. When $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}_2|_{al})$ is examined, at line 11 $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}|_{al})$ is put in $Summary$ and $Seen$.

If $\hat{\varsigma}_2$ has a predecessor $\hat{\varsigma}_3$ which is a tail call, we must show that each state $\hat{\varsigma}_4 \in CE^*_p(\hat{\varsigma}_2)$ satisfies the theorem. Wlog, we assume that $\hat{\varsigma}_4 \notin CE_p(\hat{\varsigma})$. (We have not constrained $\hat{\varsigma}_2$, so if $\hat{\varsigma}_4 \in CE_p(\hat{\varsigma})$, we have already covered this case.) Since $\hat{\varsigma}_4 \notin CE_p(\hat{\varsigma})$, $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}|_{al})$ can only be propagated in lines 33 or 41. By *IH*, $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}_3|_{al})$ was examined. There are two cases depending on which of $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}_3|_{al})$ or $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}|_{al})$ was examined first.

- $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}_3|_{al})$ was first
  By lemma 12, $|\hat{\varsigma}_2|_{al} \in succ(\hat{\varsigma}_3)$. Thus, in line 37, we put $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}_3|_{al}, |\hat{\varsigma}_2|_{al})$ in $TCallers$. When $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}|_{al})$ is examined, we follow the `else` branch at line 31. As a result, at line 33 $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}|_{al})$ is put in $Summary$ and $Seen$.
- $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}|_{al})$ was first
  Then, when $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}_3|_{al})$ is examined, $(|\hat{\varsigma}_2|_{al}, |\hat{\varsigma}|_{al})$ is in $Summary$. By lemma 12, $|\hat{\varsigma}_2|_{al} \in succ(\hat{\varsigma}_3)$. Thus, in line 41, $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}|_{al})$ is put in $Seen$. It's not put in $Summary$ because we do not want to modify $Summary$ while we're iterating over it. But lines 40 and 42 ensure that $(|\hat{\varsigma}_4|_{al}, |\hat{\varsigma}|_{al})$ will be in $Summary$ when it is examined. □