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. Unlimited 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.
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.

\[
\begin{align*}
\text{(define app} & \quad (\lambda (f \ e) (f \ e))) \\
\text{(define id} & \quad (\lambda (x) x)) \\
\text{(let*} & \quad ((n1 (app id 1)) \\
\text{\quad \quad (n2 (app id 2)))} \\
\text{\quad \quad (+ n1 n2))}
\end{align*}
\]

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:

\[
\begin{align*}
1 & \quad \text{main} \\
2 & \quad \text{app id 1} \\
3 & \quad n1 \\
4 & \quad \text{app id 2} \\
5 & \quad n2 \\
6 & \quad \text{ret} := n1+n2 \\
7 & \quad \text{main} \\
8 & \quad \text{app(f e)} \\
9 & \quad f e \\
10 & \quad \text{ret} \\
11 & \quad \text{app} \\
12 & \quad \text{id(x)} \\
13 & \quad \text{ret} := x \\
14 & \quad \text{id}
\end{align*}
\]

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].

\[
\begin{align*}
\text{(define (compose-same f x) (f (f x)))}
\end{align*}
\]

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.
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 into 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, lambda expressions and calls get labels from ULam or CLam. Labels are pairwise distinct. User lambdas take a user argument and the current continuation; lambda 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 $def _ v$, is the lambda term that contains $v$ in its list of free variables. For any term $g$, $iu _ v(g)$ is the innermost user lambda that contains $g$. Concrete syntax enclosed in $\llbracket$ 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 deep tuples are 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., $\langle 1, 3, \ldots, 47 \rangle$.

**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 if 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 $\lambda (f \ c c) (f (\lambda (v k) (c c v)) c)$, is a valid RCPS term. Terms like $\lambda (x k) (k (\lambda (y k_2) (y 123 k_2)))$ are not valid.

We can transform this term (and any CPS term) to a valid RCPS term by $\eta$-expanding to bring the free reference in operator position: $\lambda (x k) (k (\lambda (y k_2) (y 123 (\lambda (u k) (u u)))))$. 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 $(y 123 (\lambda (u k) (u u)))$, 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.

**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 (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 $\beta$ is a partial function that maps*
variables to their binding times. The variable environment \( v e \) maps
variable/time pairs to values. To find the value of a variable \( v \), we
look up the time \( v \) was put in \( \beta \), and use that to search for the
actual value in \( v e \). 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 \( U E a l \) state
over \( \langle f ( f e q ) \rangle \), we use the function \( A \) to evaluate the atomic
expressions \( f \), \( e \) and \( q \). Lambdas are paired up with \( \beta \) to become
closures, while variables are looked up in \( v e \) using \( \beta \). We add the
label \( l \) in front of the current time and transition to a \( U A p p l y \) state
(rule \([UEA]\)).

From \( U A p p l y \) to \( E a l \), we bind the formals of a procedure
\( \langle \lambda ( u, k ) \ c a l l \rangle \) to the arguments and jump to its body. The
new binding environment \( \beta' \) extends the procedure’s environment,
with \( u \) and \( k \) mapped to the current time. The new variable envi-
ronment \( v e' \) 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 \( h a l t \) to denote
the top-level continuation of a program \( p r \). The initial state \( Z ( p r ) \) is
\( \langle p r, \emptyset \rangle \), \( i n p u t \), \( h a l t \), \( \emptyset \), \( \langle \rangle \rangle \), where \( i n p u t \) is a closure of the form
\( \langle \lambda ( u, k ) \ c a l l \rangle \), \( \emptyset \). 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 envi-
nroments, a stack and a heap. We also use the stack for return-point
information; it is also an environment structure—it contains bind-
ings. CFA2 has a novel approach to variable binding: two references
to the same variable need not be looked up in the same binding en-
vironment. We split references into two categories: stack and heap
references. In direct-style, if a reference appears at the same nest-
ing level as its binder, then it is a stack reference, otherwise it is a
heap reference. For example, \( \lambda ( x ) ( \lambda ( y ) ( ( x \ ( x \ y ) ) ) ) \) 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 refer-
ences directly in CPS, without referring to the original direct-style
program. Labels can be split into disjoint sets according to the in-
nermost user lambda that contains them. For the CPS translation of
the previous program,

\[
\langle \lambda_1 ( x \ k1 ) \\
\langle k1 \ ( \lambda_2 ( y \ k2 ) \\
( x \ y \ ( \lambda_3 ( u ) \ ( x \ u \ k2 ) ) ) \rangle \rangle \rangle \]

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 \( \psi \), e.g., \( LV ( 4 ) = \{ y, k2, u \} \) and \( LV ( 6 ) =
\{ x, k1 \} \). We use this map to model stack behavior, because all
continuation lambdas that “belong” to a given user lambda \( \lambda_1 \)
get closed by extending \( \lambda_1 \)’s stack frame (cf. section 4.3). Notice
that, for any \( \psi \), \( LV ( \psi ) \) contains exactly one continuation variable.

Using \( LV \), we give the following definition.

Definition 2 (Stack and heap references).

\bullet \quad \text{Let } \psi \text{ be a call site that refers to a variable } v. \text{ The predicate } \psi \text{ (} (v, v) \text{) holds iff } v \in LV ( \psi ). \text{ We call } v \text{ a stack reference.}

\bullet \quad \text{Let } \psi \text{ be a call site that refers to a variable } v. \text{ The predicate } H ( \psi, v ) \text{ holds iff } v \notin LV ( \psi ). \text{ We call } v \text{ a heap reference.}

\bullet \quad \psi \text{ is a stack variable, written } S ( v ) \text{, iff all its references satisfy } S \psi .

\bullet \quad \psi \text{ is a heap variable, written } H ( v ) \text{, iff some of its references satisfy } H \psi .

For instance, \( S ( 5, y ) \) holds because \( y \in \{ y, k2, u \} \) and \( H ( 5, x ) \)
holds because \( x \notin \{ y, k2, u \} \).

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 ab-
stract user closure (member of the set \( UClos \)) is a set of user lamb-
das. An abstract continuation closure (member of \( CClos \)) is either
a continuation lambda or \( h a l t \). A frame is a map from variables to
abstract values, and a stack is a sequence of frames. All stack oper-
ations except \( p u s h \) 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, \([UAe]\) and \([CEA]\).

On transition from a \( U E a l \) state to a \( U A p p l y \) state (rule
\([UEA]\)), we first evaluate \( f \), \( e \) and \( q \). We evaluate atomic user
terms using \( A_u \). We non-deterministically choose one of the lamb-
das that flow to \( f \) as the operator in the \( U A p p l y \) state.1

1 An abstract execution explores one path, but the algorithm that searches
the state space considers all possible executions.
This is a non-tail call, so we do not pop.

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

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

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

\((\lambda_1(f \ cc) \ (f \ (\lambda_2(x \ k2) \ (cc \ x)) \ cc))\)

The CPS translation of its argument is

\((\lambda_3(c \ k) \ (c \ 42 \ (\lambda_4(u) \ (\text{somefunCPS} \ u \ k)))\)

The initial state \(I(pr)\) is a UApply. We abbreviate lambdas by their labels.

\((\lambda_1, \ \{\lambda_3\}, \ \text{halt}, \ (\langle\rangle, \ 0))\)

We push a frame and jump to the body of \(\lambda_1\). Since \(\lambda_3\) is a heap variable, we save the continuation and the stack in the heap. The heap \(h\) contains a single binding \(\langle\text{cc} \mapsto \langle\text{halt}, \ (\langle\rangle, \ 0)\rangle\rangle\).

\([(\text{call/cc } (\lambda (c) (\text{somefun}(c \ 42))))] \mapsto \langle\text{call/cc } (\lambda (c) (\text{somefun}(c \ 42)))) \mapsto \text{halt}\rangle, \ h\)

\(\lambda_2\) is essentially a continuation reified as a user value. We tail call to \(\lambda_3\), so we pop the stack.

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

We push a frame and jump to the body of \(\lambda_3\).

\([(\text{call/cc } (\lambda (c) (\text{somefun}(c \ 42))))] \mapsto \langle\text{call/cc } (\lambda (c) (\text{somefun}(c \ 42)))) \mapsto \text{halt}\rangle, \ h\)

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

\((\lambda_2, \ \{\lambda_2\}, \ \lambda_4, \ (\langle\text{call/cc } (\lambda (c) (\text{somefun}(c \ 42)))) \mapsto \text{halt}\rangle, \ h)\)
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 |·|_α from concrete to abstract states. Next, we show that if ς transitions to ς’ in the concrete semantics, the abstract counterpart |ς|_α of ς transitions to a state |ς’|_α which approximates |ς’|_α. Therefore, each concrete execution, i.e., sequence of states related by →, has a corresponding abstract execution that computes an approximate answer.

The map |·|_α appears in Fig. 5. The abstraction of an Eval state ς of the form ([[g_1 \ldots g_n]], β, ve, t) is an Eval state ς 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_i is the innermost user lambda that contains ψ. To reach ψ, control passed from a UApply state ς’ over λ_i. According to our stack policy, the top frame contains bindings for the formals of λ_i and any temporaries added along the path from ς’ to ς. Therefore, the domain of the top frame is a subset of LV(λ_i), i.e., a subset of LV(ψ). For each user variable u_i ∈ (LV(ψ) ∩ dom(β)), the top frame contains [u_i ← |ve(u_i, β(u_i))|_α]. Let k be the sole continuation variable in LV(ψ). If ve(k, β(k)) is halt (the return continuation is the top-level continuation), the rest of the stack is empty. If ve(k, β(k)) is ([[λ_i (u k) call]], β’), the second frame is for the user lambda in which λ_i 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 ([[λ_i (u k) call]], β) is a UApply state ς whose operator is ([[λ_i (u k) call]], β’). The stack of ς represents the environment in which the continuation argument was created, and we compute it using toStack as above.

Abstracting a CApplie 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|_α 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 \( \xi \subseteq \xi' \) is a partial order on abstract states and can be read as “\( \xi' \) is more precise than \( \xi \)” (Fig. 6). Tuples are ordered pointwise. Abstract user closures are ordered by inclusion. Two stacks are in \( \subseteq \) if they have the same length and the corresponding frames are in \( \subseteq \).

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 \( \xi \rightarrow \xi' \) and \( \xi \subseteq \xi' \) then there exists \( \xi \rightarrow \xi'' \) such that \( \xi \sim \xi'' \) and \( \xi'' \subseteq \xi'. \)

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 \( \text{append}(s', s) \).

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 algorithm to represent intra-procedural paths, hence the name.

Node 1 goes to 2, so we record the edges \((1, 1)\) and \((1, 2)\).

From 2 we call app, so we record the call \((2, 8)\) and jump to 8. In app, we find path edges \((8, 8)\) and \((8, 9)\). We find a new call \((9, 12)\) and jump to 12. Inside id, we discover the edges \((12, 12)\), \((12, 13)\) and \((12, 14)\). Edges that go from an entry to an exit, such as \((12, 14)\), 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 \((9, 12)\), so 14 returns to 10 and we find a new edge \((8, 10)\), which leads to \((8, 11)\). We record \((8, 11)\) as a summary also. From the call \((2, 8)\), we see that 11 returns to 3, so we record edges \((1, 3)\) and \((1, 4)\). Now, we have a new call to app. Reachability inside app does not depend on its calling context. From the summary \((8, 11)\), we know that 4 can reach 5, so we find \((1, 5)\). Subsequently, we find the last two path edges, which are \((1, 6)\) and \((1, 7)\).

During the search, we did two kinds of transitions. The first kind includes intra-procedural 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.

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 \( |\cdot|_{ab} \) 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 \( \text{CEval} \) state with call site \( \{(k e)^\gamma\} \) has no successor in the appendix.
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 \([\text{cc} x]\), we want to find which continuation flows to \text{cc}. We know that \(\text{def}_f(\text{cc})\) is \(\lambda_1\). By looking at the single \(\text{UApply}\) over \(\lambda_1\), we find that \text{halt} flows to \text{cc}. This suggests that, for escaping continuations, we need summaries of the form \((\tilde{c}_1, \tilde{c}_2)\) where \(\tilde{c}_2\) is an Exit-Esc over a call site \([k e]^\gamma\) and \(\tilde{c}_1\) is an entry over \(\text{def}_f(k)\).

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

If \(\tilde{c}_2\) is a call (line 14) then \(\tilde{c}_3\) is the entry of the callee, so we propagate \((\tilde{c}_1, \tilde{c}_3)\) instead of \((\tilde{c}_1, \tilde{c}_2)\) (line 16). Also, we record the call in \(\text{Callers}\). If an exit \(\tilde{c}_2\) is reachable from \(\tilde{c}_3\), it should return to the continuation born at \(\tilde{c}_2\) (line 18). The function \(\text{Update}\) is responsible for computing the return state. We find the return value \(d\) by evaluating the expression \(e_4\) passed to the continuation (lines 48-49). Since we are returning to \(\lambda_2\), we must restore the environment of its creation, which is \(I_{\tilde{c}_2}\) (possibly with stack filtering, line 50). The new state \(\tilde{c}\) is the corresponding return of \(\tilde{c}_2\), so we propagate \((\tilde{c}_1, \tilde{c})\) (lines 51-52).

If \(\tilde{c}_2\) is an Exit-Ret and \(\tilde{c}_1\) is the initial state (lines 19-20), then \(\tilde{c}_2\)'s successor is a final state (lines 53-54). If \(\tilde{c}_1\) is some other entry, we record the edge in \(\text{Summary}\) and pass the result of \(\tilde{c}_2\) to the callers of \(\tilde{c}_1\) (lines 22-23). Last, consider the case of a tail call \(\tilde{c}_4\) to \(\tilde{c}_1\) (line 24). No continuation is born at \(\tilde{c}_4\). Thus, we must find where \(\tilde{c}_1\) (the entry that led to the tail call) was called from. Then again, all calls to \(\tilde{c}_1\) 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 \(\tilde{c}_1\) to \(\tilde{c}_2\).

Let \(\tilde{c}_5\) be an Exit-Ret over a call site \([k e]^\gamma\] (line 25). Its predecessor \(\tilde{c}'\) is an entry or a \(\tilde{CApply}\). To reach \(\tilde{c}_5\), the algorithm must go through \(\tilde{c}'\). Hence, the first time the algorithm sees \(\tilde{c}_5\) is at line 7 or 13, which means that \(\tilde{c}_5\) is an entry over \(\text{UApply}(k e)^\gamma\) and \(\tilde{c}'\) is not in the \(\text{Summary}\). Thus, the test at line 26 is false. We record \(\tilde{c}_5\) in \(\text{Escapes}\). We also create summaries from entries over \(\text{def}_f(k)\) to \(\tilde{c}_5\), in order to find which continuations can flow to \(k\). We make sure to put these summaries in \(\text{Summary}\) (line 29), so that when they are examined, the test at line 26 is false.

When \(\tilde{c}_5\) is examined again, this time \((\tilde{c}_1, \tilde{c}_5)\) is in \(\text{Summary}\). If \(\tilde{c}_1\) is the initial state, \(\tilde{c}_5\) can call \text{halt} and transition to a final state (line 30). Otherwise, we look for calls to \(\tilde{c}_1\) to find continuations that can be called at \(\tilde{c}_5\) (line 32). If there are tail calls to \(\tilde{c}_1\), we propagate summaries transitively (line 33).

If \(\tilde{c}_5\) is an entry over \(\{\lambda_1(u k)\}\), its successor \(\tilde{c}_3\) is a state in the same procedure, so we propagate \((\tilde{c}_1, \tilde{c}_3)\) (lines 6-7). If \(k\) is a heap variable (lines 8-9), we put \(\tilde{c}_5\) in \(\text{EntriesEsc}\) (so that it can be found from line 29). Also, if we have seen Exit-Esc states that call \(k\), we create summaries from \(\tilde{c}_5\) to those states (line 11).

If \(\tilde{c}_5\) is a tail call (line 34), we find its successors and record the call in \(\text{TCallers}\) (lines 35-37). If a successor of \(\tilde{c}_5\) goes to an exit, we propagate a cross-procedure summary transitively (line 41). Moreover, if \(\tilde{c}_5\) is an Exit-Esc, we want to make sure that \((\tilde{c}_1, \tilde{c}_5)\) is in \(\text{Summary}\) when it is examined. We cannot call \(\text{Propagate}\) with \(\text{true}\) at line 41 because we would be mutating \(\text{Summary}\) while iterating over it. Instead, we use a temporary set which we unite with \(\text{Summary}\) after the loop (line 42).
\[\text{while } W \neq \emptyset \]
\[\text{remove } (\zeta_1, \zeta_2) \text{ from } W\]
\[\text{switch } \zeta_2 \]
\[\text{case } \zeta_2 \text{ of Entry} \]
\[\text{for the } \zeta_3 \text{ in } \text{succ}(\zeta_2), \text{Propagate}(\zeta_1, \zeta_3, \text{false})\]
\[\zeta_2 \text{ of the form } \{(\lambda (u \ k) \text{ call})\}, d, h\]
\[\text{if } H_0(k) \text{ then} \]
\[\text{insert } \zeta_2 \text{ in } \text{EntriesEsc} \]
\[\text{for each } \zeta_3 \text{ in } \text{Escapes that calls } k, \text{Propagate}(\zeta_2, \zeta_3, \text{true})\]
\[\text{case } \zeta_2 \text{ of Call} \]
\[\text{for each } \zeta_3 \text{ in } \text{succ}(\zeta_2) \]
\[\text{Propagate}(\zeta_1, \zeta_2, \text{false}) \]
\[\text{insert } (\zeta_1, \zeta_2, \zeta_3) \text{ in } \text{Callers} \]
\[\text{for each } (\zeta_3, \zeta_4) \text{ in } \text{Summary}, \text{Update}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)\]
\[\text{case } \zeta_2 \text{ of Exit-Ret} \]
\[\text{if } \zeta_1 = \mathcal{I}(pr) \text{ then } \text{Final}(\zeta_2) \]
\[\text{else} \]
\[\text{insert } (\zeta_1, \zeta_2) \text{ in } \text{Summary} \]
\[\text{for each } (\zeta_3, \zeta_4, \zeta_5) \text{ in } \text{Callers}(\zeta_1, \zeta_4, \zeta_1, \zeta_2) \]
\[\text{for each } (\zeta_3, \zeta_4, \zeta_5) \text{ in } \text{TCallers}(\zeta_3, \zeta_2, \text{false})\]
\[\text{case } \zeta_2 \text{ of Exit-Esc} \]
\[\text{if } (\zeta_1, \zeta_2) \text{ not in } \text{Summary} \text{ then} \]
\[\text{insert } \zeta_2 \text{ in } \text{Escapes} \]
\[\zeta_2 \text{ of the form } \{(k \ e)^\gamma\}, tf, h\]
\[\text{for each } \zeta_3 \text{ in } \text{EntriesEsc} \text{ over } \text{def}_{\lambda}(k), \text{Propagate}(\zeta_1, \zeta_2, \text{true})\]
\[\text{else if } \zeta_1 = \mathcal{I}(pr) \text{ then } \text{Final}(\zeta_2) \]
\[\text{else} \]
\[\text{for each } (\zeta_3, \zeta_4, \zeta_5) \text{ in } \text{Callers}(\zeta_1, \zeta_4, \zeta_1, \zeta_2) \]
\[\text{for each } (\zeta_3, \zeta_4, \zeta_5) \text{ in } \text{TCallers}(\zeta_3, \zeta_2, \text{true})\]
\[\text{case } \zeta_2 \text{ of Exit-TC} \]
\[\text{for each } \zeta_3 \text{ in } \text{succ}(\zeta_2) \]
\[\text{Propagate}(\zeta_3, \zeta_4, \text{false}) \]
\[\text{insert } (\zeta_1, \zeta_2, \zeta_3) \text{ in } \text{TCallers} \]
\[S \leftarrow \emptyset \]
\[\text{for each } (\zeta_3, \zeta_4) \text{ in } \text{Summary} \]
\[\text{insert } (\zeta_1, \zeta_4) \text{ in } S \]
\[\text{Propagate}(\zeta_1, \zeta_4, \text{false}) \]
\[\text{Summary } \leftarrow \text{Summary } \cup S \]
\[\text{Propagate}(\zeta_1, \zeta_2, \text{esc} ) \triangleq \]
\[\text{if esc then insert } (\zeta_1, \zeta_2) \text{ in } \text{Summary} \]
\[\text{if } (\zeta_1, \zeta_2) \text{ not in } \text{Seen} \text{ then insert } (\zeta_1, \zeta_2) \text{ in } \text{Seen} \text{ and } W \]
\[\text{Update}(\zeta_1, \zeta_2, \zeta_3, \zeta_4) \triangleq \]
\[\zeta_1 \text{ of the form } \{(\lambda_1(u_1 k_1) \text{ call})\}, d_1, h_1\]
\[\zeta_2 \text{ of the form } \{(f e_2 (\lambda v_2(u_2) \text{ call})^f_2)\}, tf_2, h_2\]
\[\zeta_3 \text{ of the form } \{(\lambda_3(u_3 k_3) \text{ call})\}, d_3, h_2\]
\[\zeta_4 \text{ of the form } \{(k_4 e_4)^\gamma\}, tf_4, h_4\]
\[d \leftarrow A_\lambda(e_4, \gamma_4, tf_4, h_4)\]
\[tf \leftarrow \begin{cases} 
\text{if } f = tf_2 & \leftarrow \{(f(u_3 k_3) \text{ call})\} \quad \text{Sf}(f_2,f) \\
\text{if } f & \leftarrow Hf(f_2,f) \lor \text{Lam}(f) 
\end{cases} \]
\[\zeta \leftarrow \{(\lambda v_2(u_2) \text{ call})\}, d, tf, h_4\]
\[\text{Propagate}(\zeta_1, \zeta, \text{false}) \]
\[\text{Final}(\zeta) \triangleq \]
\[\zeta \text{ of the form } \{(k e)^\gamma\}, tf, h\]
\[\text{insert } \text{(halt, } A_\lambda(e, \gamma, tf, h), \emptyset, h) \text{ in } \text{Final} \]

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 result of the algorithm to the abstract semantics: we show that if a state $\zeta$ is reachable from $\hat{I}(pr)$, then the algorithm visits $|\zeta|_ad$ (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 $\zeta$, we can uniquely identify another state $\zeta'$ as the entry of the procedure that contains $\zeta$ [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 $\zeta$ in the following path

$$\hat{I}(pr) \leadsto^* \zeta_e \leadsto^* \zeta_p \leadsto^* \zeta_p' \leadsto^* \zeta_p \leadsto^* \zeta$$

where $\zeta'$ is an Exit-Esc over $(k \ e \ v \ k)$. $\zeta_p$ and $\zeta_p'$ are entries over $\text{def}_x(k), \zeta_p$ and $\zeta_p'$ are calls. The two entries have the form

$$\zeta_p = (\text{def}_x(k), \hat{d}, \hat{e}, st, h)$$
$$\zeta_p' = (\text{def}_x(k), \hat{d'}, \hat{e'}, st', h')$$

Both $\hat{e}$ and $\hat{e'}$ can flow to $k$ and we can call either at $\hat{e}'$. If we choose to restore $\hat{e}$ and $st$ then $\hat{e}$ is in the same procedure as $\zeta_p$. If we restore $\hat{e'}$ and $st'$, $\hat{e}$ is in the same procedure as $\zeta_p'$. However, it is possible that $\hat{e} = \hat{e'}$ and $st = st'$, in which case $\zeta$ 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 $p \equiv \zeta_e \leadsto^* \zeta$ where $\zeta_e$ is an entry. We define $CE_p(\zeta)$ to be the smallest set such that:

- if $\zeta$ is an entry, $CE_p(\zeta) = \{ \zeta \}$
- if $p \equiv \zeta_e \leadsto^* \zeta, \zeta_e \leadsto^* \zeta, \zeta_e \leadsto^* \zeta$, $\zeta$ is an Exit-Esc over $(k \ e \ v \ k)$, $\zeta_1$ is an entry over $\text{def}_x(k)$, then $\zeta_1 \in CE_p(\zeta)$.
- if $p \equiv \zeta_e \leadsto^* \zeta, \zeta_e \leadsto^* \zeta, \zeta_e \leadsto^* \zeta$, $\zeta$ is neither an entry nor an Exit-Esc, $\zeta_1$ is neither an Exit-Ret nor an Exit-Esc, then $CE_p(\zeta) = CE_p(\zeta_1)$.
- if $p \equiv \zeta_e \leadsto^* \zeta, \zeta_e \leadsto^* \zeta, \zeta_e \leadsto^* \zeta, \zeta$ is a CAppl, $\zeta_2$ is a Call, then $CE_p(\zeta_1) \subseteq CE_p(\zeta)$.
- if $p \equiv \zeta_e \leadsto^* \zeta, \zeta_e \leadsto^* \zeta, \zeta_e \leadsto^* \zeta, \zeta$ is a CAppl, $\zeta_3$ is an Exit-Ret, $\zeta_3 \in CE_p(\zeta_2)$, then $CE_p(\zeta_1) \subseteq CE_p(\zeta)$.

For each state $\zeta$, we also define $CE_p(\zeta)$ to be the set of entries that can reach an entry in $CE_p(\zeta)$ through tail calls.

**Definition 6.** Let $p \equiv \zeta_e \leadsto^* \zeta$ where $\zeta_e$ is an entry. We define $CE_p(\zeta)$ to be the smallest set such that:

- $CE_p(\zeta) \subseteq CE_p(\zeta)$
- if $p \equiv \zeta_e \leadsto^* \zeta, \zeta_e \leadsto^* \zeta, \zeta_e \leadsto^* \zeta, \zeta_e \in CE_p(\zeta), \zeta_1$ is a Tail Call, then $CE_p(\zeta_1) \subseteq CE_p(\zeta)$.

Note that if $\zeta$ is an Exit-Esc over $(k \ e \ v \ k)$, a procedure that contains $\zeta$ has an entry $\zeta'$ over $\text{is}_x(\{(k \ e \ ?)\})$. Thus, $\zeta'$ is not in $CE_p(\zeta)$ because $\text{is}_x(\{(k \ e \ ?)\}) \neq \text{def}_x(k)$. For all other states, $CE_p(\zeta)$ is the set of entries of procedures that contain $\zeta$. The following lemma relates the stack of a state with the stacks of its corresponding entries.

**Lemma 7.**

Let $p \equiv \hat{I}(pr) \leadsto^* \zeta$ where $\zeta \equiv (\ldots, st, h)$.

1. If $\zeta$ is a final state then $CE_p(\zeta) = \emptyset$.
2. If $\zeta$ is an entry then $CE_p(\zeta) \neq \emptyset$. (Thus, $CE_p(\zeta) \neq \emptyset$.)
3. If $\zeta$ is an Exit-Esc then its stack is not empty and $CE_p(\zeta) \neq \emptyset$. (We do not assert anything about the stack change between a state in $CE_p(\zeta)$ and $\zeta$, it can be arbitrary.)
4. If $\zeta$ is none of the above then $CE_p(\zeta) = \emptyset$.

Let $\zeta_e \in CE_p(\zeta)/CE_p(\zeta)$ then:

- there is a frame if such that $st :: tf :: st_e$.
- there is a variable $k$ such that $tf(k') = \hat{e}$.

If $\zeta_e \in CE_p(\zeta)$ then there is a frame if such that $st :: tf :: st_e, \text{dom}(tf) \subseteq LV(l), tf(u) \subseteq d, tf(k) = \hat{e}$.

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 $p \equiv \hat{I}(pr) \leadsto^* \zeta$ then, after summarization:

- if $\zeta$ is a final state then $|\zeta|_ad \in \text{Final}$
- if $\zeta$ is not final and $\zeta \in CE_p(\zeta)$ then $(|\zeta|_ad, |\zeta|_ad) \in \text{Seen}$
- if $\zeta$ is an Exit-Ret or Exit-Esc and $\zeta \in CE_p(\zeta)$ then $(|\zeta|_ad, |\zeta|_ad) \in \text{Seen}$

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.

```plaintext
(\text{define esc (\lambda(f cc) (f (\lambda(x k) (cc x)) cc)))

(esc (\lambda(v1 k1) (v1 "foo" k1))
     (\lambda(a) (halt a)))

(esc (\lambda(v2 k2) (k2 "bar"))
     (\lambda(b) (halt b)))
```

In this program, esc is the CPS translation of call/cc. The two user functions $\lambda_1$ and $\lambda_2$ expect a reified continuation as their first argument; $\lambda_1$ uses that continuation and $\lambda_2$ does not. The abstract semantics finds that \{"foo", "bar"\} flows to $b$. However, the worker set algorithm thinks 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

$$\lambda_1 (u \ k) \ldots (\lambda_2 (u2 k2) \ldots (k \ e \ ?) \ldots) \ldots)$$

where $\lambda_1$ contains a user lambda $\lambda_2$, which in turn contains a heap reference to $k$ in operator position.

During execution, if a closure over $\lambda_2$ escapes upward, merging of continuations at \{(k \ e \ ?)\} is unavoidable. However, when $\lambda_2$ is not passed upward, the abstract semantics still merges at \{(k \ e \ ?)\}. 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 \( \lambda (x \ k1 \ k2) \ldots (k2 \ e)^2 \ldots ) \) where \( S^2(\gamma, x) \) holds. Every Exit-Ret over \( \langle k2 \ e \rangle^2 \) 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 \( \gamma \).

We have found a handler. If not, we propagate a summary \( \gamma \) and escape analysis. Consider the lambda at the beginning of this subsection. During flow analysis, we track if any closure over \( \lambda_2 \) escapes upward. We do that by checking for summaries \( \langle \xi_1, \xi_2 \rangle \), where \( \xi_1 \) is an entry over \( \lambda_1 \). If \( \lambda_2 \) is contained in a binding reachable from \( \xi_2 \) (cf [12, sec. 4.4.2]), then \( \lambda_2 \) is passed upward and we use the heap to look up at \( \langle k \ e \rangle^2 \). Otherwise, we can assume that \( \lambda_2 \) does not escape. Hence, when we see an edge \( \langle \xi_1, \xi_2 \rangle \) where \( \xi_1 \) is an entry over \( \lambda_2 \) and \( \xi_2 \) is an Exit-Esc over \( \langle k \ e \rangle^2 \), we treat it as an exception throw. We use the new transitive summaries to search deeper in the stack for a live activation of \( \lambda_1 \), 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 pushed 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 \( \langle \xi_1, \xi_2 \rangle \) where \( \xi_2 \) is same-context reachable from \( \xi_1 \). 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 (2, 3), (2, 6), (5, 7). 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’ \( \Delta \)CFA [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 \( \Delta \)CFA-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
    https://github.com/mozilla/doctorjs
    http://www.python.org/dev/peps/pep-0255


A.

We assume that CFA2 works on an alphetized program, i.e., a program where all variables have distinct names. Thus, if an alphetized program contains a term \( \lambda \alpha (v_1 \ldots v_n) \text{call} \), 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 \( \varsigma \) has the form \( (\ldots, v, t) \).

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

**Proof.** We show that the lemma holds for the initial state \( \mathcal{I}(pr) \).

Then, for each transition \( \varsigma \rightarrow \varsigma' \), we assume that \( \varsigma \) satisfies the lemma and show that \( \varsigma' \) also satisfies it.

- **\( \mathcal{I}(pr) \)** is a UApply of the form \((pr, \emptyset), (\lambda \alpha, \emptyset), \text{halt}, \emptyset, ()\).
  Since \( \varsigma \) is empty, (1) trivially holds. Also, both closures have an empty environment so (3) holds.
- The \([UEA]\) transition is:
  \[
  \begin{aligned}
  &\{ (f \in \mathcal{Q}) \}, \beta, v, t \rightarrow (\text{proc}, d, c, v, t) \\
  &\text{proc} = A(f, \beta, v) \\
  &d = A(e, \beta, v) \\
  &c = A(q, \beta, v)
  \end{aligned}
  \]
  The \( v \) doesn’t change in the transition, so (1) holds for \( \varsigma' \).
  The operator is a closure of the form \((\lambda \alpha, \beta')\). We must show that \( \text{dom}(\beta') \cap BV(\lambda) = \emptyset \) if \( \text{Lam}(f) \), then \( \lambda \alpha = f \) and \( \beta' = \beta \). Also, we know \( \text{dom}(\beta) \cap BV(\{ f \}) = \emptyset \) so by (4) we get the desired result because \( \varsigma \) satisfies (1). Similarly for \( \beta' \) and \( c \).
- The \([\text{UAE}]\) transition is:
  \[
  \begin{aligned}
  &\{ (f \in \mathcal{Q}) \}, \beta, v, t \rightarrow (\text{call}, \beta', v, t) \\
  &\text{proc} \equiv (\{ (\lambda \alpha (u \text{ call}) \}, \beta) \\
  &\beta' = \beta[u \rightarrow t[k \rightarrow t]] \\
  &v' = v' = v'[u, t \rightarrow d][k, t \rightarrow c]
  \end{aligned}
  \]
  To show (1) for \( v' \), it suffices to show that \( d \) and \( c \) don’t violate the property. The user argument \( d \) is of the form \( (\lambda \alpha, \beta_1) \). Since \( \varsigma \) satisfies (3), we know \( \text{dom}(\beta_1) \cap BV(\text{lam}1) = \emptyset \), which is the desired result. Similarly for \( \beta' \). Also, we must show that \( \varsigma' \) satisfies (2). We know \( \{ u, k \} \cap BV(\text{call}) = \emptyset \) because the program is alphetized. Also, from property (3) for \( \varsigma \) we know \( \text{dom}(\beta) \cap BV(\{ (\lambda \alpha (u \text{ call}) \}) = \emptyset \), which implies \( \text{dom}(\beta) \cap BV(\text{call}) = \emptyset \). We must show \( \text{dom}(\beta') \cap BV(\text{call}) = \emptyset \) \( \iff \) \( \text{dom}(\beta) \cup \{ u, k \} \cap BV(\text{call}) = \emptyset \) \( \iff \) \( \text{dom}(\beta') \cap BV(\text{call}) = \emptyset \) \( \iff \) \( \emptyset \cup \emptyset = \emptyset \).
  - Similarly for the other two transitions.

**Theorem 10 (Simulation).** If \( \varsigma \rightarrow \varsigma' \) and \( |\varsigma|_{\alpha} \subseteq \varsigma \), then there exists \( \varsigma'' \) such that \( \varsigma \rightarrow \varsigma'' \) and \( |\varsigma''|_{\alpha} \subseteq \varsigma'' \).

**Proof.** By cases on the concrete transition.
form \( ((q \ e)^\gamma)^\varnothing, st, h) \), where
\[
\begin{align*}
|ve|_\alpha & \sqsubseteq h, \quad ts \sqsubseteq st \\
\end{align*}
\]
(5)
The abstract transition is
\[
\begin{align*}
((q \ e)^\gamma)^\varnothing, st, h) \leadsto (\hat{c}, \hat{d}, st', h) \\
\end{align*}
\]
\(d = \mathcal{A}_\varnothing(e, \gamma, st, h)\)
\(\hat{c}, st' \in \{(q, st)\} \cup \{(st, pop(st))\} \cup \{\hat{c}; \hat{d}, st'\}
\]
\(h(q) = \mathcal{H}(\gamma, q)\)
Let \(ts'\) be the stack of \(|\beta'|_\alpha\). We must show \(|\beta'|_\alpha \sqsubseteq \beta'\), i.e.,
\[
\begin{align*}
|\text{proc}|_\alpha & = \hat{c} \\
|d|_\alpha & \sqsubseteq \hat{d} \\
\end{align*}
\]
(6)
(7)
(8)
Showing (7) is simple, by cases on \(e\).
We will show (6) and (8) simultaneously, by cases on \(q\):
• \(\mathcal{Lam}(q)\)
  Then, \(\text{proc} = (q, \beta)\) and \(\hat{c} = q\), which imply (6).
  Also, \(sl = st\), so (8) follows from \(\text{ts} \sqsubseteq st\)
  \(\leftarrow \text{toStack}(LV(L(q)), \beta, ve) \subseteq st\)
  \(\leftarrow \text{toStack}(LV(\gamma), \beta, ve) \subseteq st\)
  \(\leftarrow (5)\)
• \(\mathcal{S}(\gamma, q)\) and \(\text{proc} = ve(q, \beta(q)) = (\lambda:\beta, 1)\)
  For (6), it suffices to show \(\hat{c} = \lambda\)
  \(\leftarrow st(q) = \lambda\)
  \(\leftarrow \text{ts}(q) = \lambda\)
  \(\leftarrow q \in LV(\gamma)\)
  which holds because \(\mathcal{S}(\gamma, q)\).
  For (8), it suffices to show
  \(\text{toStack}(LV(\lambda:\beta), \beta, ve) \subseteq pop(st)\)
  \(\leftarrow \text{pop}(ts) \subseteq pop(st)\)
  \(\leftarrow (5)\)
• \(\mathcal{H}(\gamma, q)\) and \(\text{proc} = ve(q, \beta(q)) = \text{halt}\)
  Similar to the previous case.
• \(\mathcal{H}(\gamma, q)\) and \(\text{proc} = ve(q, \beta(q)) = (\lambda:\beta, 1)\)
  In this case, \(|\text{proc}|_\alpha = \lambda\)
  and \(\text{toStack}(LV(\lambda:\beta), \beta, ve) \in |ve|_\alpha(q)\).
  By (5), there exists a pair \((\lambda:\beta, \beta', ve) \in \text{ts}\).
  By picking this pair for \(\beta'\), we get (6) and (8) because
  \(\text{ts}' = \text{toStack}(LV(\lambda:\beta), \beta, ve)\).
• \(\mathcal{H}(\gamma, q)\) and \(\text{proc} = ve(q, \beta(q)) = \text{halt}\)
  Similar to the previous case.

\[\square\]

Lemma 11 (Same-level reachability).
Let \(p \equiv \hat{I}(pr) \leadsto^\ast \hat{c}\) where \(\hat{c} \equiv (\ldots, st, h)\).
1. If \(\hat{c}\) is a final state then \(CE_p(\hat{c}) = \emptyset\).
2. If \(\hat{c}\) is an entry then \(CE_p(\hat{c}) \neq \emptyset\).
3. If \(\hat{c}\) is an Exit-Esc then its stack is not empty and \(CE_p(\hat{c}) \neq \emptyset\).
4. If \(\hat{c}\) is none of the above then \(CE_p(\hat{c}) \neq \emptyset\).

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

Now we prove the lemma considering only the two cases for \(S\).
• For each \(\hat{c}_1\) in \(S\), \(\hat{I}(pr) \in CE_p(\hat{c}_1)\).
• For each \(\hat{c}_2\) in \(S\), \(\hat{I}(pr) \notin CE_p(\hat{c}_2)\).
• For each $\zeta_i$ in $S$, $\hat{I}(pr) \notin CE_P(\zeta_i)$. Let $\tilde{\zeta}_i \neq \hat{I}(pr)$ be the earliest state in $CE_P(\zeta_i)$. Then, its predecessor $\zeta_i$ is $\zeta_i$ and its stack is $st$. Thus, $p$ has the form $\hat{I}(pr) \sim^+ \zeta_i \sim^+ \tilde{\zeta}_i \sim^+ \zeta$. By $IH$ for $\hat{I}(pr)$, we get that the continuation argument of $\tilde{\zeta}_i$ is $\zeta_i$ and its stack is $st$. Then, by rule $[UEA]$, we get that $\hat{c}$ is the continuation lambda appearing at the call site of $\zeta_i$. Thus, $\zeta_i$ is not a final state, so we must show that $CE_P(\zeta_i) \neq \emptyset$. By the fourth item of def. 5, $CE_P(\zeta_i) \subseteq CE_P(\zeta)$. But by $IH$ for $\hat{I}(pr)$, we get that $CE_P(\zeta_i) \neq \emptyset$. Thus, $CE_P(\zeta) \neq \emptyset$. We now proceed to prove the remaining obligations for the states in $CE_P(\zeta)$.

Let $\tilde{c}_i \in CE_P(\zeta)$, of the form $[(\lambda t. (u k')) call]_t, \tilde{c}_{st}, st_{se}, st_{te}, h_s)$. $\tilde{c}_i$ has the form $[(\langle e_1, e_2 q \rangle^t), st_3, h_3]$ where $q = \tilde{c}$. By $IH$ for $\hat{I}(pr) \Rightarrow^* \tilde{\zeta}_i$, we get that $st_3 = tf :: st_{se}, dom(tf) \subseteq LV(I), tf(u) \subseteq d_c, tf(k') = \tilde{c}_i, t'_i \in LL(I)$.

$\tilde{c}_i$ has the form $(\lambda a. d_2, \tilde{c}, s, h, t')$, where $st = tf' :: st_{se} and tf' = \begin{cases} \text{Lam}(v_1) \rightarrow H_2(t'_i, e_1) \end{cases}$. We can see that the stack has the appropriate form: $tf'(k') = tf(k') = \tilde{c}$ and $tf'(u) \subseteq tf(u) \subseteq d_c$. Also, $\tilde{c}$ is a lambda appearing at $t'$, so $\hat{L}(\tilde{c}) \in LL(I)$. For the case where $\zeta_i \in CE_P(\zeta_i) \setminus CE_P(\tilde{\zeta}_i)$, the proof is similar and simpler.

$\zeta$ is a $CA$Apply and $\zeta$ is an Exit-Ret.

The two states have the form:

$\zeta = (\tilde{c}, d, st, h)$

$\zeta = (\langle (k e)^t \rangle, st', h)$

By $IH$ for $\hat{I}(pr) \Rightarrow^* \tilde{\zeta}_i$, we get $CE_P(\zeta_i) \neq \emptyset, CE_P(\tilde{\zeta}_i) \neq \emptyset, CE_P(\tilde{\zeta}_i) \neq \emptyset$. Also, by $IH$ and rule $[CEA]$, $st' = tf :: st_{se}, tf(k) = \tilde{c}$.

$\hat{I}(pr) \in CE_P(\tilde{\zeta}_i)$

By $IH$ for $\hat{I}(pr) \Rightarrow^* \tilde{\zeta}_i$, we get $\tilde{c} = halt, st = \emptyset$. Thus, $\zeta$ is a final state, so we must show $CE_P(\zeta) = \emptyset$. Assume that $CE_P(\zeta) \neq \emptyset$. This can only happen if the fifth item of def. 5 applies. In this case, $p$ has the form $\hat{I}(pr) \Rightarrow^* \zeta_i \sim^+ \zeta_j \sim^+ \zeta \sim^+ \zeta$. By $IH$ for $\hat{I}(pr) \Rightarrow^* \tilde{\zeta}_i$, we get that the continuation argument of $\tilde{\zeta}_i$ is $\zeta_i$ and by rule $[UEA]$ we get that the stack is $\zeta_i$. Thus, $\zeta$ is not a final state, so we must show that $CE_P(\zeta) \neq \emptyset$. By the fifth item of def. 5 we know that $CE_P(\zeta) \subseteq CE_P(\zeta_i)$ and by $IH$ we know that $CE_P(\tilde{\zeta}_i) \neq \emptyset$. To show that the stack has the desirable properties, we work in the same way as in the case where $\zeta$ is a $CA$Apply and $\zeta$ is an Exit-Esc.

$\zeta$ is none of the above.

In this case, $\zeta$ is one of: $UEval_{inner} CEval, Exit-Ret, CAApply whose predecessor is an inner CEval$. The path can be decomposed as $\hat{I}(pr) \Rightarrow^* \tilde{\zeta}_i \sim^+ \zeta$. By $IH$, $CE_P(\tilde{\zeta}_i) \neq \emptyset$, and by the third item of def. 5, $CE_P(\tilde{\zeta}_i) \neq \emptyset$. It’s simple to show that the stack of $\tilde{\zeta}_i$ has the desired properties by assuming that the stack of $\tilde{\zeta}_i$ has them.

### Lemma 12 (Local simulation).

If $\zeta \sim^+ \tilde{\zeta}_i and \ succ(|\tilde{\zeta}_i|) \neq \emptyset$ then $|\tilde{\zeta}_i| \in \ succ(|\zeta|)$.

### Theorem 13 (Soundness).

If $p \equiv \hat{I}(pr) \Rightarrow^* \zeta$ then, after summarization:

- If $\tilde{c}$ is a final state then $|\tilde{\zeta}_i| \in Final$
- If $\zeta$ is not final and $\zeta \in CE_P(\zeta)$ then $(|\zeta|_a, |\tilde{\zeta}_i|_a) \in \efinal$
- If $\tilde{c}$ is an Exit-Ret or Exit-Esc and $\zeta \in CE_P(\zeta)$ then $(|\zeta|_a, |\tilde{\zeta}_i|_a) \in \efinal$
- If $\tilde{c}$ is an Exit-Esc and $\zeta \in CE_P(\zeta)$ then $(|\zeta|_a, |\tilde{\zeta}_i|_a)$ 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 $\hat{I}(pr) \Rightarrow^* \zeta$. We take cases on $\zeta$.

$\zeta$ is an entry.

Then, $CE_P(\zeta) = \{\zeta\}$. Also, $\zeta$ is a call or a tail call.

By lemma 11, $CE_P(\zeta) \neq \emptyset$. Let $\tilde{c}_i \in CE_P(\zeta)$. Then, $p$ can be decomposed as $\hat{I}(pr) \Rightarrow^* \zeta_i \sim^+ \tilde{\zeta}_i \sim^+ \zeta$. By $IH$, $(|\tilde{\zeta}_i|_a, |\zeta|_a)$ was put in $\efinal$ at some point during the execution, so it was also put in $W$ and examined. By lemma 12, $(|\tilde{\zeta}_i|_a) \in \ succ(|\zeta|_a)$ so in line 16 or 36 $(|\tilde{\zeta}_i|_a, |\zeta|_a)$ will be propagated.

$\zeta$ is a $CA$Apply and $\zeta$ is an Exit-Esc.

The two states have the form:

$\zeta = (\tilde{c}, d, st, h)$

$\zeta = (\langle (k e)^t \rangle, st', h)$

By lemma 11, $CE_P(\zeta) \neq \emptyset$. Since $\tilde{c}, st \in h(k)$, there is a state $\tilde{c}_i \in CE_P(\zeta)$ of the form:

$\tilde{c}_i = (def_{\tilde{c}_i}(d_1, \tilde{c}, st, h_1))$

Also, by $IH$, $\tilde{c}$ was examined first. By lemma 12, $|\tilde{\zeta}_i| \in \ succ(|\zeta|)$.

By in line 16 or 36 $(|\tilde{\zeta}_i|_a, |\zeta|_a)$ will be propagated.

$\tilde{\zeta}_i \neq \hat{I}(pr)$ is the earliest state in $CE_P(\zeta')$. The predecessor $\tilde{\zeta}_i$ of $\tilde{\zeta}_i$ is a call. By $IH$ for $\hat{I}(pr) \Rightarrow^* \tilde{\zeta}_i$, we get that the continuation argument of $\tilde{\zeta}_i$ is $\zeta$ and its stack is $st$. By rule $[UEA]$, we get that $\hat{c}$ is the continuation lambda appearing at the call site of $\zeta_i$. Thus, $\zeta$ is not a final state, so we must show that $CE_P(\zeta) \neq \emptyset$. By the fifth item of def. 5 we know that $CE_P(\tilde{\zeta}_i) \subseteq CE_P(\tilde{\zeta}_i)$ and by $IH$ we know that $CE_P(\tilde{\zeta}_i) \neq \emptyset$. To show that the stack has the desirable properties, we work in the same way as in the case where $\tilde{\zeta}_i$ is a $CA$Apply and $\tilde{\zeta}_i$ is an Exit-Esc.
is in Summary when it is examined, so the test at line 26 is false. Also, $\xi_2 \neq \xi$ (pr) so the test at line 30 is false as well. Since $\langle \xi_4aal, \xi_5aal, \xi_2aal \rangle$ is in Callers, at line 32 we call $\text{Update}(\xi_4aal, \xi_5aal, \xi_2aal, \xi')$. We must show that the state $\xi$ constructed by $\text{Update}$ is the same as $\xi_aal$.

By lemma 11 for $\hat{I}(pr) \sim^+ \xi'$, we get that $st'$ is not empty. It is easy to see that the user value passed at $\xi$, which is $A_\alpha(e, \gamma, |st'|aal; |h|aal)$, is equal to $A_\alpha(e, \gamma, st', h)$.

By lemma 11 for $\hat{I}(pr) \sim^+ \xi$, the stack of $\xi$ is not empty. Thus, $\xi_3$ has the form $\langle (e_1 e_2 q) \rangle, tf :: st_3, h_3 \rangle$ where $q \equiv \hat{e}$. Let $\text{ulam}$ be the function applied at $\xi_2$. Then, by rule [UEA], the stack $st$ of $\xi_2$ is

$$st = \begin{cases} tf :: st_3 & \text{Lam}\gamma(e_1) \lor H_\gamma(l, e_1) \\ tf[e_1 \mapsto \{\text{ulam}\}] :: st_3 & S_\gamma(l, e_1) \end{cases}$$

But then, $|st|aal$ is equal to the frame constructed at line 50. Therefore, $\xi \equiv \xi_aal$.

Assume that $\langle |\xi_2|aal, |\xi'|aal \rangle$ was examined first. In this case, when $\langle |\xi_4|aal, |\xi_5|aal \rangle$ is examined, we call $\text{Update}$ at line 18. The proof is similar.

• $\langle |\xi_2|aal, |\xi_2|aal \rangle$ was first
  When $\langle |\xi_1|aal, |\xi|aal \rangle$ is examined, the test at line 26 is true. Also, $\langle |\xi|aal, |\xi|aal \rangle$ is in EntriesEsc, it was put at line 10 when $\langle |\xi_2|aal, |\xi_2|aal \rangle$ was examined. Thus, at line 29, $\langle |\xi_2|aal, |\xi|aal \rangle$ is put in Summary and Seen.

• $\langle |\xi|aal, |\xi|aal \rangle$ was first
  At line 27, $|\xi|aal$ was put in Escapes. When $\langle |\xi_2|aal, |\xi_2|aal \rangle$ is examined, at line 11 $\langle |\xi_2|aal, |\xi|aal \rangle$ is put in Summary and Seen.

If $\xi_2$ has a predecessor $\xi_3$ which is a tail call, we must show that each state $\xi_4 \in CE_p(\xi_2)$ satisfies the theorem. Wlog, we assume that $\xi_4 \notin CE_p(\xi)$. (We have not constrained $\xi_4$, so if $\xi_4 \notin CE_p(\xi)$, we have already covered this case.) Since $\xi_4 \notin CE_p(\xi)$, $(|\xi_4|aal, |\xi|aal)$ can only be propagated in lines 33 or 41. By IH, $(|\xi_4|aal, |\xi_3|aal)$ was examined. There are two cases depending on which of $\langle |\xi_4|aal, |\xi_3|aal \rangle$ or $\langle |\xi_2|aal, |\xi|aal \rangle$ was examined first.

• $\langle |\xi_4|aal, |\xi_3|aal \rangle$ was first
  By lemma 12, $\langle |\xi_2|aal \in succ(\xi_3) \rangle$. Thus, in line 37, we put $\langle |\xi_4|aal, |\xi_3|aal, |\xi_2|aal \rangle$ in TCallers. When $\langle |\xi_2|aal, |\xi|aal \rangle$ is examined, we follow the else branch at line 31. As a result, at line 33 $\langle |\xi_4|aal, |\xi|aal \rangle$ is put in Summary and Seen.

• $\langle |\xi_2|aal, |\xi|aal \rangle$ was first
  Then, when $\langle |\xi_4|aal, |\xi_3|aal \rangle$ is examined, $\langle |\xi_2|aal, |\xi|aal \rangle$ is in Summary. By lemma 12, $\langle |\xi_2|aal \in succ(\xi_3) \rangle$. Thus, in line 41, $\langle |\xi_2|aal, |\xi|aal \rangle$ 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 $\langle |\xi_4|aal, |\xi|aal \rangle$ will be in Summary when it is examined.