CFA2: Pushdown Flow Analysis for Higher-Order Languages

A dissertation presented by

Dimitrios Vardoulakis

to the Faculty of the Graduate School
of the College of Computer and Information Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Northeastern University
Boston, Massachusetts

August 20, 2012

Abstract

In a higher-order language, the dominant control-flow mechanism is function call and return. Most higher-order flow analyses do not handle call and return well: they remember only a bounded number of pending calls because they approximate programs as finite-state machines. Call/return mismatch introduces precision-degrading spurious execution paths and increases the analysis time.

We present flow analyses that provide unbounded call/return matching in a general setting: our analyses apply to typed and untyped languages, with first-class functions, side effects, tail calls and first-class control. This is made possible by several individual techniques. We generalize Sharir and Pnueli's summarization technique to handle expressive control constructs, such as tail calls and first-class continuations. We propose a syntactic classification of variable references that allows precise lookups for non-escaping references and falls back to a conservative approximation for references captured in closures. We show how to structure a flow analysis like a traditional interpreter based on big-step semantics. With this formulation, expressions use the analysis results of their subexpressions directly, which minimizes caching and makes the analysis faster. We present experimental results from two implementations for Scheme and JavaScript, which show that our analyses are precise and fast in practice.

Acknowledgments

Thanks to my parents, Manolis and Litsa, and my sisters, Maria and Theodosia, for their love and support. Thanks to my loving wife Laura for being in my life. Meeting her was the best side effect of my Ph.D.

I have been very lucky to have Olin as my advisor. He shared his vision for programming languages with me, believed in me, and gave me interesting problems to solve. He was also the source of many hilarious stories.

Thanks to my thesis committee members, Alex Aiken, Matthias Felleisen and Mitch Wand for their careful comments on my dissertation. Thanks to all members of the Programming Research Lab at Northeastern; I learned a tremendous amount during my time there. Thanks to Pete Manolios and Agnes Chan for their advice and encouragement. Danny Dubé visited Northeastern for a semester, showed interest in my work and found the first program that triggers the exponential behavior in CFA2. The front-office staff at the College of Computer Science, especially Bryan Lackaye and Rachel Kalweit, always made bureaucracy disappear.

Thanks to Dave Herman and Mozilla for giving me the opportunity to apply my work to JavaScript. The summer of 2010 in Mountain View was one of the best in my life. Mozilla also graciously supported my research for two years.

I was in Boston during the Christmas break of 2008. At that time, I had settled on what the abstract semantics of CFA2 would look like, but had not figured out how to get around the infinite state space. On December 29, while googling for related papers, I ran into Swarat Chaudhuri's POPL 2008 paper. His beautiful description of summarization in the introduction made me realize that I could use this technique for CFA2.

Nikos Papaspyrou, my undergraduate advisor, introduced me to the world of programming language research. My uncle Yiannis tutored me in math when I was in high school. He was the best teacher I ever had. Vassilis

Koutavas and I shared an apartment in Boston for three years. Thanks for the great company. Last but not least, thanks to my good friends in Athens and Heraklion, who make every trip home worthwhile.

Notes to the reader

A reader with the background of a beginning graduate student in programming languages should be able to follow the technical material in this dissertation. In particular, we assume good understanding of basic computer-science concepts such as finite-state machines, pushdown automata and asymptotic complexity. We also assume familiarity with the λ-calculus and operational semantics. Finally, some high-level intuition about program analysis and its applications is helpful, but we do not assume any knowledge of higher-order flow analysis.

We develop the theory of CFA2 in the untyped λ-calculus. However, we take the liberty to add primitive data types such as numbers and strings in the examples to keep them short and clear.

Parts of this dissertation are based on material from the following papers:

CFA2: a Context-Free Approach to Control-Flow Analysis
Dimitrios Vardoulakis and Olin Shivers
European Symposium on Programming (ESOP), pages 570–589, 2010

Ordering Multiple Continuations on the Stack
Dimitrios Vardoulakis and Olin Shivers
Partial Evaluation and Program Manipulation (PEPM), pages 13–22, 2011

CFA2: a Context-Free Approach to Control-Flow Analysis
Dimitrios Vardoulakis and Olin Shivers
Logical Methods in Computer Science (LMCS), 7(2:3), 2011

Pushdown Flow Analysis of First-Class Control
Dimitrios Vardoulakis and Olin Shivers
International Conf. on Functional Programming (ICFP), pages 69–80, 2011

Contents

CHAPTER 1

Introduction

Flow analysis seeks to predict the behavior of a program without running it. It reveals information about the control flow and the data flow of a program, which can be used for a wide array of purposes.

- Compilers use flow analysis to perform powerful optimizations such as inlining, constant propagation, common-subexpression elimination, loop-invariant code motion, register allocation, *etc*.
- Flow analysis can find errors that are not detected by the type system, such as null dereference and out-of-bounds array indices. In untyped languages, it can find type-related errors.
- Flow analysis can prove the absence of certain classes of errors, *e.g.*, show that a program will not throw an exception or that casts will not fail.
- Last, modern code editors utilize flow analysis to assist with development: refactoring, code completion, *etc*.

Flow analysis works by simulating the execution of a program using a simplified, approximate semantics. There are several reasons for this. First, the properties of interest are often undecidable. Also, the analysis does not know the inputs that will be supplied to the program at runtime; it must produce valid results for all possible inputs. Last, the program may consist of several compilation units, some of which are not available to the analysis.

In the present work, we study flow analysis for higher-order languages. A higher-order language allows computation to be packaged up as a value. The primary way to treat computation as data is through first-class functions.

Higher-order languages provide expressive features that allow the concise and elegant description of computations. Unsurprisingly, the expressiveness of higher-order languages makes flow analysis difficult.

- First and foremost, control flow in higher-order programs is not syntactically apparent. For example, in functional programs we see call sites with a variable in operator position, *e.g.*, `(x e)`. At runtime, one or more functions will flow to `x` and get called—data flow induces control flow. Consequently, higher-order flow analyses must reason about control flow and data flow in tandem.
- Higher-order languages often lack static types. They may also allow overloading of operators and automatic conversion of values from one type to another at runtime. Flow analyses must cope with such flexibility.
- Some higher-order languages provide powerful control constructs such as tail calls, exceptions, generators and first-class continuations. These constructs make it hard to predict the target of a control-flow transfer at compile time.

Most flow analyses approximate programs as *finite graphs* of abstract machine states [61, 3, 68, 75, 44].[1] Each node in such a graph represents a program point plus some amount of information about the environment and the calling context. The analysis considers every path from the start to the end node of the graph to be a possible execution of the program. Therefore, execution traces can be described by a *regular language*.

This approximation does not model function call and return well; it permits paths that do not properly match calls with returns. During the analysis, a function may be called from one program point and return to a different one. Call/return mismatch causes spurious flow of data, which decreases precision.

The effects of call/return mismatch in a first-order flow analysis can perhaps be mitigated by the fact that most control flow in first-order languages happens with conditionals and loops, not with function calls. However, function (and method) calls are central in higher-order programs, so modeling them faithfully is crucial for a flow analysis.

Pushdown analyses can match an unbounded number of calls and returns. These analyses approximate programs as *pushdown automata* or equivalent machines. By pushing return points on the stack of the automaton, it is possible to always return to the correct place. Pushdown analyses have long been used for first-order languages [59, 52, 21, 6, 2]. However, existing

[1] Citations are listed in chronological order.

pushdown analyses for higher-order languages [45, 50, 66, 31, 17] are not sufficiently general, *e.g.*, some apply to typed languages only, and none handles first-class control constructs.

Contributions

In this dissertation, we propose general techniques for pushdown analyses of higher-order programs. Our main contributions are:

- We generalize the summarization technique of Sharir and Pnueli [59] to higher-order languages and introduce a new kind of summary edge to handle tail calls.
- We develop a variant of continuation-passing style that allows effective reasoning about the stack in the presence of first-class control. We use this variant to perform pushdown analysis of languages with first-class control.
- We identify a class of variable references that can and should be handled precisely by a flow analysis. Most references in typical programs belong in this class, so handling them well results in significant precision gains.
- We propose structuring a flow analysis like a traditional interpreter, *i.e.*, as a collection of mutually recursive functions, using big-step semantics. With this formulation, it becomes possible to minimize caching of abstract states, which results in a lightweight and fast analysis.

Outline

The rest of this dissertation is organized as follows.

- Chapter 2 provides the necessary background on abstract interpretation and continuation-passing style (CPS).
- Chapter 3 describes how finite-state analyses work, through the lens of Shivers's kCFA analysis [61]. We show that most limitations of finite-state analyses are direct consequences of call/return mismatch.
- In chapter 4, we present our CFA2 analysis. We formulate CFA2 as an abstract interpretation of programs in a CPS λ-calculus. CFA2 classifies every variable reference as either a stack or a heap reference. Stack references can be handled precisely because they cannot be captured in heap-allocated closures. We provide examples which show that CFA2

overcomes the limitations of finite-state analyses.

In CFA2, abstract states have a stack of unbounded height. Therefore, the abstract state space is infinite. We describe an algorithm based on summarization that explores the state space without loss in precision. We state theorems that establish the correctness of CFA2. Last, we briefly discuss the complexity of the analysis.

- In chapter 5, we present Restricted CPS and use it to generalize CFA2 to handle first-class control.
- Chapter 6 presents an alternative approach to higher-order pushdown analysis, called Big CFA2. Big CFA2 shares some traits with CFA2: the split between stack and heap references and the use of summaries. We formulate Big CFA2 as an abstract interpretation of a direct-style λ-calculus and show how to implement it efficiently. We also present extensions to the algorithm for mutation and exceptions.
- To provide support for the practical applicability of our ideas, we implemented DoctorJS, a static-analysis tool for the full JavaScript language based on Big CFA2. Chapter 7 discusses the trade-offs involved in designing a static analysis for JavaScript.
- In chapter 8, we present experimental results. We implemented CFA2 for a subset of Scheme and compared its precision to kCFA. We found that CFA2 is more precise and usually visits a smaller state space. In addition, we used DoctorJS for type inference of JavaScript programs and to study the interaction between browser add-ons and web pages. The results indicate that DoctorJS is precise, reasonably fast, and scales to large, real-world programs.
- We present related work in chapter 9, discuss directions for future research in chapter 10 and conclude in chapter 11.
- Appendix A includes proofs of the theorems that establish the correctness of CFA2.

CHAPTER 2

Preliminaries

This chapter introduces the formal machinery that we use to develop our analyses: abstract interpretation and continuation-passing style. Abstract interpretation is a method for program analysis created by Cousot and Cousot [11, 12]. Continuation-passing style is used as an intermediate representation of programs in several functional-language compilers [67, 38, 4, 34].

2.1 Abstract interpretation

This section describes how to formulate a static analysis as an abstract interpretation. We want to analyze programs written in some Turing-complete language L. The semantics of L programs can be described by a transition relation $\Rightarrow$ between program states. We write $\varsigma \Rightarrow \varsigma'$ to mean that a state ς transitions to ς'. This semantics is called the *concrete* semantics.

The role of the concrete semantics is to evaluate programs; it does not collect any analysis-related information, *e.g.*, it does not record which functions can be applied at a particular call site. For this reason, we adapt the concrete semantics to derive the so-called *non-standard* concrete semantics, which performs the analysis of interest without loss in precision (see sec. 2.2.3 for an example). The non-standard semantics is incomputable. Therefore, we must find some relation $\rightsquigarrow$ that is an *abstraction*, *i.e.*, a conservative approximation, of the non-standard semantics.

Let $\rightarrow$ be the non-standard concrete transition relation. What does it mean for $\rightsquigarrow$ to approximate $\rightarrow$? First, there is a map $|\cdot|$ from concrete to abstract states. Second, there is an ordering relation $\sqsubseteq$ on abstract states (reflexive, transitive and antisymmetric). We write $\hat{\varsigma} \sqsubseteq \hat{\varsigma}'$ to mean that $\hat{\varsigma}'$ is more approximate than $\hat{\varsigma}$. (By convention, abstract elements have the same

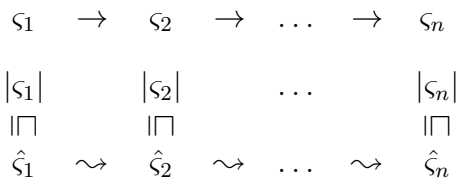


Figure 2.1: Relating concrete and abstract executions

names as their concrete counterparts, but with a $\widehat{\ }$ symbol over them.) Then, we prove the following theorem.

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

We say that the abstract semantics *simulates* the concrete semantics. From this theorem, it follows that each concrete execution, *i.e.*, sequence of states related by $\to$, has a corresponding abstract execution that computes an approximate answer (fig. 2.1). Intuitively, the abstract semantics does not miss any flows of the concrete semantics, but it may add extra flows that never happen.

2.2 The elements of continuation-passing style

Before we can perform flow analysis, we need a representation of programs that facilitates static reasoning. In flow analysis of λ-calculus-based languages, a program is usually turned to an intermediate form where all subexpressions are named before it is analyzed. This form can be Continuation-Passing Style (CPS), A-Normal Form [22], or ordinary direct-style λ-calculus where each expression has a unique label [48, ch. 3]. An analysis using one form can be changed to use another form without much effort.

This work uses CPS. We opted for CPS because it reifies continuations as λ terms, which makes it suitable for reasoning about non-local control operators such as exceptions and first-class continuations. For example, we can define `call/cc` as a function in CPS; we do not need a special primitive operator to express it.

This section explains the basics of CPS. In CPS, every function has an additional formal parameter, the *continuation*, which represents the "rest of the computation." To return a value v to its context, a function calls its continuation with v. At every function call, we pass a continuation as an extra

argument. Hence the name Continuation-Passing Style. The process that turns a program from direct style to CPS is called the *CPS transformation*.

Let's see an example. We define a function that computes the discriminant of a quadratic equation and call it.

```
(define (discriminant a b c)
  (- (* b b) (* 4 a c)))

(discriminant 1 5 4)
```

The corresponding CPS program is

```
(define (discriminant a b c k)
  (%* b b
      (λ (b2) (%* 4 a c
                  (λ (ac4) (%- b2 ac4 k))))))

(discriminant 1 5 4 halt)
```

`Discriminant` takes a continuation parameter `k`. Assuming left-to-right evaluation, `(* b b)` happens first, followed by `(* 4 a c)`. The `%` sign signifies that the primitive operators are now CPS functions: `%*` takes some numbers, multiplies them and passes the product to its continuation (similarly for `%-`). The last argument of each call is a continuation. The program terminates by calling its top-level continuation $halt$.

Notice that the operator and the arguments of every call are *atomic* expressions (primitives, constants, variables or lambdas), never calls. Therefore, in CPS call-by-value and call-by-name evaluation coincide. We must take the evaluation order into account during the CPS transformation.

2.2.1 Partitioned CPS

Compilers that use CPS 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.

The functions in the direct-style program become user functions in CPS; the lambdas introduced by the transformation are continuation functions. User functions take some user arguments and one continuation argument; continuations take one user argument. Each call is classified as user or con-

	Functions	Variables	Calls
User	`discriminant` `%*` `%-`	`a` `b` `c` `b2` `ac4`	`(%* b b ...)` `(%* 4 a c ...)` `(%- b2 ac4 k)` `(discriminant 1 5 4` *halt*`)`
Continuation	`(λ(b2) ...)` `(λ(ac4) ...)` *halt*	`k`	

Table 2.1: User/continuation partitioning for `discriminant`

tinuation according to its operator. The runtime semantics respects the static partitioning: user (*resp.* continuation) values flow only to user (*resp.* continuation) variables. Table 2.1 shows the partitioning for the `discriminant` example. (Even though there are no continuation call sites, the continuations are called implicitly by the primitive functions.)

2.2.2 Syntax and notation

We develop the theory of CFA2 in a CPS λ-calculus with the user/continuation distinction. The syntax appears in fig. 2.2. User and continuation elements get labels from the disjoint sets $ULab$ and $CLab$.

Without loss of generality, we assume that all variables in a program have distinct names and all terms are uniquely labeled. For a particular program pr, we write $\mathcal{L}(g)$ for the label of term g in pr and $\mathcal{T}(\psi)$ for the term labeled with ψ in pr. $FV(g)$ returns the free variables of term g. The defining lambda of x, written $def_\lambda(x)$, is the lambda that contains x in its list of formals. For a 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.*, $3 :: \langle 1, 3, -47 \rangle$. We write $\pi_i(\langle e_1, \ldots, e_n \rangle)$ to project the i^{th} element of a tuple $\langle e_1, \ldots, e_n \rangle$.

$x \in Var$	$= UVar + CVar$	Program variables
$u \in UVar$	= a set of identifiers	User variables
$k \in CVar$	= a set of identifiers	Continuation variables
$\psi \in Lab$	$= ULab + CLab$	Program labels
$l \in ULab$	= a set of labels	Labels of user elements
$\gamma \in CLab$	= a set of labels	Labels of continuation elements
$lam \in Lam$	$= ULam + CLam$	Lambda terms
$ulam \in ULam$	$::= (\lambda_l (u\ k)\ call)$	User lambdas
$clam \in CLam$	$::= (\lambda_\gamma (u)\ call)$	Continuation lambdas
$call \in Call$	$= UCall + CCall$	Call expressions
$UCall$	$::= (f\ e\ q)^l$	Call to a user function
$CCall$	$::= (q\ e)^\gamma$	Call to a continuation function
$g \in Exp$	$= UExp + CExp$	Atomic expressions
$f, e \in UExp$	$= ULam + UVar$	Atomic user expressions
$q \in CExp$	$= CLam + CVar$	Atomic continuation expressions
$pr \in Program$	$= ULam$	Program to be analyzed

Figure 2.2: Partitioned CPS

2.2.3 Concrete semantics

The execution of programs in Partitioned CPS can be described by a state-transition system (fig. 2.3). 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. $Eval$ states are classified as user or continuation depending on their call site, $Apply$ states depending on their operator.

We use environments (partial functions from variables to values) for variable binding. On entering the body of a function (rules [UAE] and [CAE]) we extend the environment with bindings for the formal parameters.

The function $\mathcal{A}$ evaluates atomic expressions, *i.e.*, lambda terms and variables. The initial execution state $\mathcal{I}(pr)$ is a $UApply$ state of the form $((pr, \emptyset), input, halt)$, where $input$ is some user closure $(ulam, \emptyset)$.

Unbounded environment chains Not surprisingly, the concrete state space is infinite. (If it were finite we could enumerate all states in finite time and solve the halting problem, which is impossible since the λ-calculus is Turing complete.) Environments allow the creation of infinite structure because

$$
\begin{aligned}
\varsigma \in \mathit{State} &= \mathit{Eval} + \mathit{Apply} \\
\mathit{Eval} &= \mathit{UEval} + \mathit{CEval} \\
\mathit{UEval} &= \mathit{UCall} \times \mathit{Env} \\
\mathit{CEval} &= \mathit{CCall} \times \mathit{Env} \\
\mathit{Apply} &= \mathit{UApply} + \mathit{CApply} \\
\mathit{UApply} &= \mathit{UProc} \times \mathit{UProc} \times \mathit{CProc} \\
\mathit{CApply} &= \mathit{CProc} \times \mathit{UProc} \\
\mathit{Proc} &= \mathit{UProc} + \mathit{CProc} \\
d \in \mathit{UProc} &= \mathit{ULam} \times \mathit{Env} \\
c \in \mathit{CProc} &= (\mathit{CLam} \times \mathit{Env}) + \{\mathit{halt}\} \\
\rho \in \mathit{Env} &= \mathit{Var} \rightharpoonup \mathit{Proc}
\end{aligned}
$$

(a) Domains

$$
\mathcal{A}(g,\rho) \triangleq \begin{cases} (g,\rho) & \mathit{Lam}_?(g) \\ \rho(g) & \mathit{Var}_?(g) \end{cases}
$$

$$[\text{UEA}] \quad (\llbracket (f\ e\ q)^l \rrbracket, \rho) \Rightarrow (\mathcal{A}(f,\rho), \mathcal{A}(e,\rho), \mathcal{A}(q,\rho))$$

$$[\text{UAE}] \quad (\langle \llbracket (\lambda_l (u\ k)\ \mathit{call}) \rrbracket, \rho \rangle, d, c) \Rightarrow (\mathit{call}, \rho[u \mapsto d,\ k \mapsto c])$$

$$[\text{CEA}] \quad (\llbracket (q\ e)^\gamma \rrbracket, \rho) \Rightarrow (\mathcal{A}(q,\rho), \mathcal{A}(e,\rho))$$

$$[\text{CAE}] \quad (\langle \llbracket (\lambda_\gamma (u)\ \mathit{call}) \rrbracket, \rho \rangle, d) \Rightarrow (\mathit{call}, \rho[u \mapsto d])$$

(b) Semantics

Figure 2.3: Concrete semantics of Partitioned CPS

they contain closures, which in turn contain other environments. An environment chain can contain multiple closures over the same lambda. Some of these closures must be "muddled together" during flow analysis.

Non-standard semantics Shivers proposed adding a level of indirection in environments [61]. *Binding* environments map variables to addresses and *variable* environments map variable-address pairs to values. To bind a variable x to a value v, we allocate a fresh address a, bind x to a in the binding environment and (x, a) to v in the variable environment.

The non-standard semantics appears in fig. 2.4. The last component of each state is a *time*, which is a sequence of call sites. Eval-to-Apply tran-

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

(a) Domains

$$
\mathcal{A}(g, \rho, ve) \triangleq \begin{cases} (g, \rho) & \mathit{Lam}_?(g) \\ ve(g, \rho(g)) & \mathit{Var}_?(g) \end{cases}
$$

[UEA] $(\llbracket (f\ e\ q)^l \rrbracket, \rho, ve, t) \to (\mathcal{A}(f, \rho, ve), \mathcal{A}(e, \rho, ve), \mathcal{A}(q, \rho, ve), ve, l :: t)$

[UAE] $(\langle \llbracket (\lambda_l (u\ k)\ \mathit{call}) \rrbracket, \rho \rangle, d, c, ve, t) \to (\mathit{call}, \rho', ve', t)$
$\rho' = \rho[u \mapsto t,\ k \mapsto t\,]$
$ve' = ve[(u, t) \mapsto d,\ (k, t) \mapsto c\,]$

[CEA] $(\llbracket (q\ e)^\gamma \rrbracket, \rho, ve, t) \to (\mathcal{A}(q, \rho, ve), \mathcal{A}(e, \rho, ve), ve, \gamma :: t)$

[CAE] $(\langle \llbracket (\lambda_\gamma (u)\ \mathit{call}) \rrbracket, \rho \rangle, d, ve, t) \to (\mathit{call}, \rho', ve', t)$
$\rho' = \rho[u \mapsto t\,]$
$ve' = ve[(u, t) \mapsto d\,]$

(b) Semantics

Figure 2.4: Non-standard concrete semantics of Partitioned CPS

sitions 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. We use times as addresses for variable binding. We write $t_1 \sqsubset t_2$ when t_1 is a proper suffix of t_2, *i.e.*, when t_1 is an earlier time than t_2 ($\sqsubseteq$ to include equality).

$\mathcal{A}$ takes the variable environment ve as an extra argument. To find the value of a variable x, we look up the time when x was put in the binding environment ρ and use that to search for the actual value in ve.

The initial state $\mathcal{I}(pr)$ is a $UApply$ of the form $((pr, \emptyset), input, halt, \emptyset, \langle\rangle)$. The initial time is the empty sequence of call sites. The final state (if any) is a $CApply$ of the form $(halt, d, ve, t)$, where d is the result of evaluating the program. Any *non-final* state that does not have a successor is a *stuck* state, *e.g.*, $([\![(f\ e\ q)^l]\!], \emptyset, \emptyset, t)$ where f, e and q are variables.

The non-standard semantics performs flow analysis alongside evaluation. The analysis results are recorded in the variable environment ve. Every time a value flows to a variable, *i.e.*, in the $Apply$-to-$Eval$ transitions, the semantics records the flow in ve. We can query ve to answer flow-analysis questions. For example, the following set contains all closures that can flow to a variable x.

$$\{\, ve(x,t) \;:\; \forall t.(x,t) \in \mathrm{dom}(ve)\}$$

In the rest of this dissertation, we do not use the standard concrete semantics; the analyses in chapters 3, 4 and 5 are abstractions of the non-standard semantics. Thus, for the sake of brevity, when we refer to the "concrete semantics" without using a qualifier, we mean the non-standard concrete semantics.

2.2.4 Preliminary lemmas

Most states in $State$ are unreachable when we evaluate a program pr under the concrete semantics. We are only interested in states that can be realized when starting from $\mathcal{I}(pr)$. In this section, we show some basic properties of reachable states.

Definition 1. Let $S \subseteq Var$, ρ, ve satisfy the property $\mathcal{P}_{\mathrm{bind}}(S, \rho, ve)$ iff $S \subseteq \mathrm{dom}(\rho)$ and $\forall x \in \mathrm{dom}(\rho).\ (x, \rho(x)) \in \mathrm{dom}(ve)$.

Definition 1 states that all variables in a set S are bound in ρ and the variable-time pairs are bound to some closure in ve.

Lemma 2. Let ς be a reachable state of the form $(\ldots, ve, t)$. Then,

1. For every $(lam, \rho) \in \mathrm{range}(ve)$, $\mathcal{P}_{\mathrm{bind}}(FV(lam), \rho, ve)$ holds.
2. If ς is an *Eval* state $(call, \rho, ve, t)$, $\mathcal{P}_{\mathrm{bind}}(FV(call), \rho, ve)$ holds.
3. If ς is an *Apply* and (lam, ρ) is a closure in operator or argument position, $\mathcal{P}_{\mathrm{bind}}(FV(lam), \rho, ve)$ holds.

Lemma 2 states that when we look up a free variable, it is always bound to some closure; variable lookups never fail. This lemma ensures that reachable states do not get stuck. Thus, the evaluation of a program either reaches a final state or diverges.

Definition 3. For any term g, the map $BV(g)$ returns the variables bound by g or by lambdas which are subterms of g. It has a simple inductive definition:
$BV([\![(\lambda_\psi (x_1 \ldots x_n)\, call)]\!]) = \{x_1, \ldots, x_n\} \cup BV(call)$
$BV([\![(g_1 \ldots g_n)^\psi]\!]) = BV(g_1) \cup \cdots \cup BV(g_n)$
$BV(x) = \emptyset$

We assume that all variables in a program have distinct names. Thus, if $(\lambda_\psi (x_1 \ldots x_n)\, call)$ is a term in such a program, we know that no other lambda binds variables with names $x_1, \ldots, x_n$. (During the analysis, we do not rename any variables.) The following lemma is a simple consequence of using unique variable names.

Lemma 4. Let ς be a reachable state of the form $(\ldots, ve, t)$. Then,

1. For any closure $(lam, \rho) \in \mathrm{range}(ve)$, it holds that $\mathrm{dom}(\rho) \cap BV(lam) = \emptyset$.
2. If ς is an *Eval* state $(call, \rho, ve, t)$, then $\mathrm{dom}(\rho) \cap BV(call) = \emptyset$.
3. If ς is an *Apply* state, any closure (lam, ρ) in operator or argument position satisfies $\mathrm{dom}(\rho) \cap BV(lam) = \emptyset$.

Every lambda lam appears textually in operator or argument position at some call site ψ. When the execution reaches an *Eval* state of the form $(\mathcal{T}(\psi), \rho, ve, t)$, we pair up lam with ρ to create a closure. Let $x \in FV(lam)$. In a well-formed state, we expect that x was entered in ρ before the creation of the closure, not at some future time (!) In other words, $\rho(x) \sqsubseteq t$ should hold. The next lemma ensures the well-formedness of times.

Definition 5. Let ρ, ve satisfy the property $\mathcal{P}_{\text{time}}(\rho, ve)$ iff for every $x \in \text{dom}(\rho)$ where $ve(x, \rho(x)) = (lam, \rho')$ and $t \in \text{range}(\rho')$, we know $t \sqsubset \rho(x)$.

Lemma 6 (Temporal consistency)**.**
Let ς be a reachable state of the form $(\ldots, ve, t)$. Then,

1. For every $(lam, \rho) \in \text{range}(ve)$, $\mathcal{P}_{\text{time}}(\rho, ve)$ holds.
2. If ς is an $Eval$ state $(call, \rho, ve, t)$, $\mathcal{P}_{\text{time}}(\rho, ve)$ holds. Also, for any time t' in ρ or in ve, $t' \sqsubseteq t$.
3. If ς is an $Apply$ and (lam, ρ) is a closure in operator or argument position, $\mathcal{P}_{\text{time}}(\rho, ve)$ holds. Also, for any time t' in ρ or in ve, $t' \sqsubset t$.

To prove these lemmas, we show that they hold for $\mathcal{I}(pr)$ and that every transition preserves their truth. We include a proof of lemma 4 in appendix A. The others are similar.

CHAPTER 3

The kCFA analysis

The most well-known flow analysis for languages with first-class functions is Shivers's kCFA [60, 61, 62], which is a finite-state analysis. In this chapter, we use kCFA as the vehicle for understanding how all finite-state analyses work. We describe the specification of kCFA and how it can be implemented using a simple workset algorithm. We state the results that establish the soundness of kCFA (without proof). Last, we discuss the limitations of kCFA and other finite-state models and identify call/return mismatch as the main reason for these limitations.

3.1 Introduction to kCFA

This section presents the main ideas of kCFA using examples. The discussion is informal to get the intuition across, so we overlook the subtle differences between variants of the analysis (widening: none *vs.* per state *vs.* global, reachability or not, *etc.*). The next section shows the formal semantics.

kCFA is not a single analysis, but a family of analyses, parameterized over a natural number k. The parameter k is a bounded representation of the calling context, like Sharir and Pnueli's *call strings* [59]. For $k = 0$, we get a *monovariant* analysis, *i.e.*, invocations of a function from different calling contexts are not distinguished. For $k > 0$, the analysis is *polyvariant*: it creates contexts of length k that represent the last k function activations and it distinguishes invocations of a function that happen in different contexts.

Consider the following program, which defines the apply and identity functions, then binds `n1` to 1 and `n2` to 2 and adds them. At program point `(+ n1 n2)`, there is only one possible value for `n1` and only one possible value for `n2`; we would like a static analysis to be able to find these values.

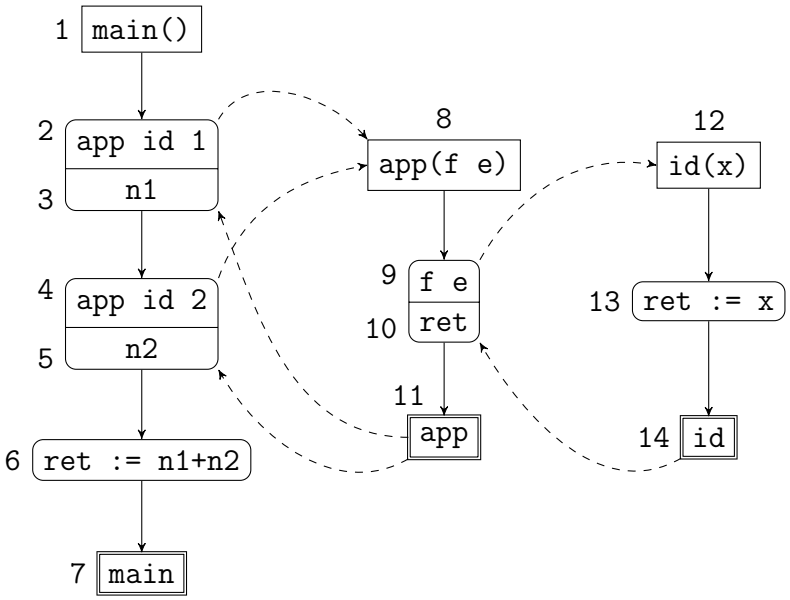


Figure 3.1: 0CFA control-flow graph

```
(define app (λ (f e) (f e)))
(define id  (λ (x) x))

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

0CFA produces the control-flow graph in fig. 3.1. 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.

0CFA, like all monovariant analyses, treats interprocedural control flow (dashed arrows: calls and returns) in the same way as intraprocedural control flow (solid arrows). It considers all paths in the graph to be valid executions and it uses a single global environment for variable binding. For these

Address	Value	Address	Value	Address	Value
`n1`	1, 2	`f`	`id`	`x`	1, 2
`n2`	1, 2	`e`	1, 2	`ret_id`	1, 2
`ret_main`	2, 3, 4	`ret_app`	1, 2		

(a) 0CFA

Address	Value	Address	Value	Address	Value
`n1`	1, 2	`f`$\langle 2 \rangle$	`id`	`x`$\langle 9 \rangle$	1, 2
`n2`	1, 2	`e`$\langle 2 \rangle$	1	`ret_id`$\langle 9 \rangle$	1, 2
`ret_main`	2, 3, 4	`ret_app`$\langle 2 \rangle$	1, 2		
		`f`$\langle 4 \rangle$	`id`		
		`e`$\langle 4 \rangle$	2		
		`ret_app`$\langle 4 \rangle$	1, 2		

(b) 1CFA

Address	Value	Address	Value	Address	Value
`n1`	1	`f`$\langle 2 \rangle$	`id`	`x`$\langle 9, 2 \rangle$	1
`n2`	2	`e`$\langle 2 \rangle$	1	`ret_id`$\langle 9, 2 \rangle$	1
`ret_main`	3	`ret_app`$\langle 2 \rangle$	1	`x`$\langle 9, 4 \rangle$	2
		`f`$\langle 4 \rangle$	`id`	`ret_id`$\langle 9, 4 \rangle$	2
		`e`$\langle 4 \rangle$	2		
		`ret_app`$\langle 4 \rangle$	2		

(c) 2CFA

Table 3.1: kCFA results for the example program

reasons, it cannot distinguish between different calls to the same function. 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 `(+ n1 n2)`, 0CFA determines that each variable can be bound to either 1 or 2. Table 3.1a shows the results of 0CFA.

In section 2.2.3, we mentioned that flow analyses have to approximate the environment chain in some way. 0CFA approximates all closures over a lambda lam in the environment chain with a single closure. In fact, it uses a single closure for all closures over lam that are created during the course of the analysis, not just for the ones that can be live in the same environment chain.

We can increase precision by increasing k. For $k = 1$, the analysis distin-

guishes calls to a function f that happen at different program points. However, if f is called from a program point in function g and g is called from two different places, 1CFA merges these two calls to f. Table 3.1b shows the results of 1CFA. Note that 1CFA finds that both 1 and 2 flow to `n1` and `n2`, because it merges the two calls to `id`.

By increasing k once more, we get the precise result (table 3.1c). Note that the call strings are of length up to 2, not always 2. For example, if a function is called from the top level, its call string is 1. Intuitively, for $k > 0$, kCFA constructs a graph where each program point appears as many times as the different call strings for that program point.

3.2 The semantics of kCFA

The kCFA semantics is an abstraction of the semantics of section 2.2.3. We can get a computable analysis by making Time finite. We fix some number k and remember only the k most recent labels of a call string. For example, if $k = 2$, call strings $\langle 12, 79, 3, 46\rangle$ and $\langle 12, 79, 5\rangle$ are both approximated by $\langle 12, 79\rangle$. As a result, the analysis merges distinct bindings. For a variable x and functions f_1, f_2, the bindings

$$\begin{matrix}[(x, \langle 12, 79, 3, 46\rangle) \mapsto f_1\,] \\ [(x, \langle 12, 79, 5\rangle) \mapsto f_2\,]\end{matrix} \quad \text{are merged to} \quad [(x, \langle 12, 79\rangle) \mapsto \{f_1, f_2\}\,].$$

Variables are bound to sets of closures instead of a single closure. Fig. 3.2a shows the abstract domains of kCFA. Note that the state space is *finite*.

The transition rules of the abstract semantics appear in fig. 3.2b. They are similar to the rules of fig. 2.4. The main difference is that the abstract semantics is *non-deterministic*. In the $\widehat{\mathit{Eval}}$-to-$\widehat{\mathit{Apply}}$ transitions (rules $[\widehat{\mathrm{UEA}}]$, $[\widehat{\mathrm{CEA}}]$), the operator evaluates to a set of closures; we pick one non-deterministically and jump to it .

Since Time is finite, when we add a binding $[(x, \hat{t}) \mapsto \hat{v}\,]$ to $\widehat{ve}$ we do not know if $\hat{t}$ is fresh, so we *join* the new binding with any existing ones. The join operation $\sqcup$ is defined as:

$$(\widehat{ve} \sqcup [(x, \hat{t}) \mapsto \hat{v}\,])(y, \hat{t}') \triangleq \begin{cases} \widehat{ve}(x, \hat{t}) \cup \hat{v} & (y, \hat{t}') = (x, \hat{t}) \\ \widehat{ve}(y, \hat{t}') & \text{o/w} \end{cases}$$

$$
\begin{array}{rcl}
\hat{\varsigma} \in \widehat{State} & = & \widehat{Eval} + \widehat{Apply} \\
\widehat{Eval} & = & \widehat{UEval} + \widehat{CEval} \\
\widehat{UEval} & = & UCall \times \widehat{BEnv} \times \widehat{VEnv} \times \widehat{Time} \\
\widehat{CEval} & = & CCall \times \widehat{BEnv} \times \widehat{VEnv} \times \widehat{Time} \\
\widehat{Apply} & = & \widehat{UApply} + \widehat{CApply} \\
\widehat{UApply} & = & \widehat{UProc} \times \widehat{UVal} \times \widehat{CVal} \times \widehat{VEnv} \times \widehat{Time} \\
\widehat{CApply} & = & \widehat{CProc} \times \widehat{UVal} \times \widehat{VEnv} \times \widehat{Time} \\
\widehat{UProc} & = & ULam \times \widehat{BEnv} \\
\widehat{CProc} & = & (CLam \times \widehat{BEnv}) + \{halt\} \\
\hat{v} \in \widehat{Val} & = & \widehat{UVal} + \widehat{CVal} \\
\hat{d} \in \widehat{UVal} & = & Pow(\widehat{UProc}) \\
\hat{c} \in \widehat{CVal} & = & Pow(\widehat{CProc}) \\
\hat{\rho} \in \widehat{BEnv} & = & Var \rightharpoonup \widehat{Time} \\
\widehat{ve} \in \widehat{VEnv} & = & Var \times \widehat{Time} \rightharpoonup \widehat{Val} \\
\hat{t} \in \widehat{Time} & = & Lab^k
\end{array}
$$

(a) Domains

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

$$
\begin{array}{ll}
[\widehat{\text{UEA}}] & (\llbracket (f\ e\ q)^l \rrbracket, \hat{\rho}, \widehat{ve}, \hat{t}) \rightsquigarrow (\widehat{proc}, \hat{\mathcal{A}}(e, \hat{\rho}, \widehat{ve}), \hat{\mathcal{A}}(q, \hat{\rho}, \widehat{ve}), \widehat{ve}, \lceil l :: \hat{t} \rceil^k) \\
& \widehat{proc} \in \hat{\mathcal{A}}(f, \hat{\rho}, \widehat{ve}) \\
\\
[\widehat{\text{UAE}}] & (\langle \llbracket (\lambda_l (u\ k)\ call) \rrbracket, \hat{\rho} \rangle, \hat{d}, \hat{c}, \widehat{ve}, \hat{t}) \rightsquigarrow (call, \hat{\rho}', \widehat{ve}', \hat{t}) \\
& \hat{\rho}' = \hat{\rho}[u \mapsto \hat{t},\ k \mapsto \hat{t}\,] \\
& \widehat{ve}' = \widehat{ve} \sqcup [(u, \hat{t}) \mapsto \hat{d},\ (k, \hat{t}) \mapsto \hat{c}\,] \\
\\
[\widehat{\text{CEA}}] & (\llbracket (q\ e)^\gamma \rrbracket, \hat{\rho}, \widehat{ve}, \hat{t}) \rightsquigarrow (\widehat{proc}, \hat{\mathcal{A}}(e, \hat{\rho}, \widehat{ve}), \widehat{ve}, \lceil \gamma :: \hat{t} \rceil^k) \\
& \widehat{proc} \in \hat{\mathcal{A}}(q, \hat{\rho}, \widehat{ve}) \\
\\
[\widehat{\text{CAE}}] & (\langle \llbracket (\lambda_\gamma (u)\ call) \rrbracket, \hat{\rho} \rangle, \hat{d}, \widehat{ve}, \hat{t}) \rightsquigarrow (call, \hat{\rho}', \widehat{ve}', \hat{t}) \\
& \hat{\rho}' = \hat{\rho}[u \mapsto \hat{t}\,] \\
& \widehat{ve}' = \widehat{ve} \sqcup [(u, \hat{t}) \mapsto \hat{d}\,]
\end{array}
$$

(b) Semantics

Figure 3.2: Abstract semantics of kCFA

```
1  Seen ← {Î(pr)}
2  W ← {Î(pr)}
3  while W ≠ ∅
4    remove a state ς̂ from W
5    foreach ς̂₂ such that ς̂ ⇝ ς̂₂
6      if ς̂₂ ∉ Seen then
7        insert ς̂₂ in Seen and W
```

Figure 3.3: Workset algorithm for kCFA

When we increment time, its length may exceed k, so we use the $\lceil\cdot\rceil^k$ function to keep only the k most recent labels. The initial state $\hat{\mathcal{I}}(pr)$ is $((pr, \emptyset), \{input\}, \{halt\}, \emptyset, \langle\rangle)$.

3.2.1 Workset algorithm

Let $\mathcal{RS}$ be the set of abstract states that are reachable from $\hat{\mathcal{I}}(pr)$.

$$\mathcal{RS} = \{\, \hat{\varsigma} \;:\; \hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}\}$$

We can visualize $\mathcal{RS}$ as a directed graph, whose nodes are abstract states, and there is an edge from a node $\hat{\varsigma}_1$ to a node $\hat{\varsigma}_2$ iff $\hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}_2$. This graph is the result of the kCFA analysis. Any algorithm for graph reachability, such as breadth-first search or depth-first search, can compute this graph.

Fig. 3.3 shows a simple workset algorithm that computes the graph. The exploration pattern is left unspecified; one can make it breadth-first by making the workset a queue or depth-first by making it a stack. $\widehat{State}$ is finite and we mark states as seen when we put them in W. Therefore, the algorithm terminates. Note that even though $\widehat{Eval}$ states non-deterministically transition to one of their possible successors, the algorithm explores all successors of a state to ensure soundness. When it finishes, $Seen$ is equal to $\mathcal{RS}$.

3.2.2 Correctness

We show that the abstract semantics of kCFA safely approximates the concrete semantics, using the methodology of section 2.1.

The abstraction function $|\cdot|$ maps each concrete state to an abstract state (fig. 3.4). The first four equations show how to abstract states. The call site of an $Eval$ stays the same and we apply $|\cdot|$ to ρ, ve and t. Similarly, when

$$|(call, \rho, ve, t)| = (call, |\rho|, |ve|, |t|)$$

$$|(\langle ulam, \rho\rangle, d, c, ve, t)| = (\langle ulam, |\rho|\rangle, |d|, |c|, |ve|, |t|)$$

$$|(\langle clam, \rho\rangle, d, ve, t)| = (\langle clam, |\rho|\rangle, |d|, |ve|, |t|)$$

$$|(halt, d, ve, t)| = (halt, |d|, |ve|, |t|)$$

$$|(lam, \rho)| = \{(lam, |\rho|)\}$$

$$|halt| = \{halt\}$$

$$|t| = \lceil t \rceil^k$$

$$|\rho| = \lambda\ x.\ |\rho(x)|$$

$$|ve| = \lambda\ x, \hat{t}.\ \bigcup_{|t|=\hat{t}} |ve(x,t)|$$

Figure 3.4: Abstraction function

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

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

$$\hat{\rho}_1 \sqsubseteq \hat{\rho}_2 \quad \text{iff} \quad \hat{\rho}_1 = \hat{\rho}_2 \restriction \mathrm{dom}(\hat{\rho}_1)$$

$$\widehat{ve}_1 \sqsubseteq \widehat{ve}_2 \quad \text{iff} \quad \forall (x, \hat{t}) \in \mathrm{dom}(\widehat{ve}_1).\ \widehat{ve}_1(x, \hat{t}) \sqsubseteq \widehat{ve}_2(x, \hat{t})$$

Figure 3.5: Ordering relation on abstract states

abstracting an $Apply$, the lambda of the operator stays the same and we apply $|\cdot|$ to the environment and the other components of the state.

Closures and $halt$ abstract to singleton sets because abstract values are sets of closures. Times abstract to call strings of length k. We abstract binding environments pointwise. To abstract a variable environment, we must join several bindings together; when we look up the value of $(x, \hat{t})$ in $|ve|$, there may be many times t in ve that abstract to $\hat{t}$. Therefore, for each such t, we find the closure $ve(x, t)$, abstract it and include it in the result.

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. 3.5). Tuples are ordered pointwise. Abstract

values are ordered by inclusion. A binding environment $\hat{\rho}_2$ is more approximate than $\hat{\rho}_1$ iff they agree on the domain of $\hat{\rho}_1$ ($\restriction$ is the function-restriction operator). A variable environment $\widehat{ve}_2$ is more approximate than $\widehat{ve}_1$ iff for every binding $((x, \hat{t}), \hat{v})$ in $\widehat{ve}_1$, $\widehat{ve}_2$ has a more approximate binding. The following examples illustrate the use of $\sqsubseteq$.

$$\begin{array}{rcl} halt & \sqsubseteq & halt \\ [\![(f\ e\ q)^l]\!] & \sqsubseteq & [\![(f\ e\ q)^l]\!] \\ \{(x, \langle 7, 19\rangle)\} & \sqsubseteq & \{(x, \langle 7, 19\rangle),\quad (y, \langle 3\rangle)\} \\ \{(x, \langle 7, 19\rangle),\quad (y, \langle 3\rangle)\} & \not\sqsubseteq & \{(x, \langle 7, 19\rangle),\quad (y, \langle 3, 5\rangle)\} \end{array}$$

We omit the proof of the simulation theorem because the soundness of kCFA is well known. The interested reader can find a proof in Might's dissertation [42].

3.3 Limitations of finite-state analyses

kCFA considers every path in the abstract-state graph to be a possible execution of the program. Thus, executions are strings in a *regular* language. However, the execution traces that properly match calls with returns are strings in a *context-free* language. Approximating this control flow with regular-language techniques permits execution paths that break call/return nesting. Call/return mismatch affects the analysis in several ways.

3.3.1 Imprecise dataflow information

In kCFA, a function may be called from one program point but return to a different one, which results in spurious flow of data. In section 3.1, we saw an example for 0CFA. The analysis cannot match the call site and the return point of `app` and this causes 2 to flow to `n1` and 1 to flow to `n2`. Similar examples can be written for any k.

The `app/id` example is an atypical program, but it illustrates an issue found in all real-world programs. It is common to have a few functions that are called from tens or hundreds of program points. If an analysis is not effective at distinguishing different calling contexts, widely used functions pollute the analysis results.

3.3.2 Inability to calculate stack change

Besides the spurious flow of data, call/return mismatch results in spurious control flow. As a consequence, we cannot accurately calculate stack changes between program points. The `app/id` example is a straight-line program, but according to 0CFA, it has a loop (there is a path from 4 to 3). Recursive programs make stack-change calculation even harder.

Some optimizations, however, 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.
- Often, continuations captured by `call/cc` do not escape upward. In this case, we do not need to copy the stack into the heap.

Such optimizations are performed more effectively with pushdown analyses.

3.3.3 Sensitivity to syntax changes

A finite-state analysis approximates the program stack, whose size is unbounded, with a finite number of contexts. As a result, seemingly innocent syntactic transformations that preserve the meaning of a program may confuse the analysis because they "eat up" context.

In the following example, 1CFA can find that each variable can be bound only to a single number.

```
(let* ((id (λ (x) x))
       (n1 (id 1))
       (n2 (id 2)))
  (+ n1 n2))
```

If we η-expand `id`, 1CFA can no longer find the precise answer.

```
(let* ((id (λ (y) ((λ (x) x) y)))
       (n1 (id 1))
       (n2 (id 2)))
  (+ n1 n2))
```

By η-expanding `id` repeatedly, we can consume as many call sites as we want, so for any k we can construct an example that tricks kCFA.

Polymorphic splitting [75], which is a finite-state analysis that creates a separate context for each occurrence of a let-bound variable, would be precise in this example. But if we lambda-bind `id`, polymorphic splitting merges the results of the two calls.

```
((λ (id)
    (let* ((n1 (id 1))
           (n2 (id 2)))
      (+ n1 n2)))
 (λ (x) x))
```

3.3.4 The environment problem and fake rebinding

In higher-order languages, many bindings of the same variable can be simultaneously live. Determining at compile time whether two references to some variable will be bound in the same runtime environment is referred to as the *environment problem* [61, 42]. Consider the following program:

```
(let ((f (λ (x thunk)
            (if (number? x)
                (thunk)
                (λ1() x)))))
  (f 0 (f "foo" "bar")))
```

In the inner call to `f`, `x` is bound to `"foo"` and λ_1 is returned. We call `f` again; this time, `x` is 0, so we jump through `(thunk)` to λ_1, and reference `x`, which, despite the just-completed test, is not a number: it is the string `"foo"`.

Thus, during static analysis, it is generally unsafe to assume that a reference has some property just because an earlier reference had that property. This has an unfortunate consequence: when two references are bound in the same environment, kCFA does not detect it, and it allows paths in which the references are bound to different abstract values. We call this phenomenon *fake rebinding*.

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

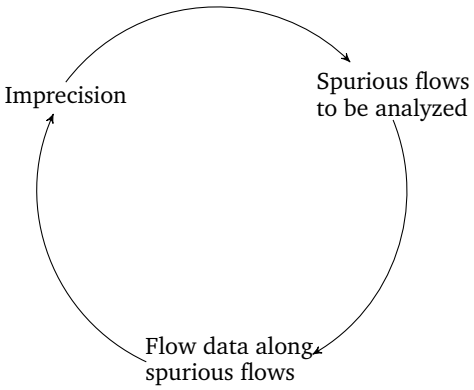


Figure 3.6: The vicious cycle of approximation

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`, kCFA may call one closure at the inner call site and a different closure at the outer call site.

3.3.5 Polyvariant versions of kCFA are intractable

kCFA for $k > 0$ is an expensive analysis, both in theory [70] and in practice [64]. Counterintuitively, imprecision in higher-order flow analyses can increase their running time: imprecision induces spurious paths, along which the analysis must flow data, thus creating further spurious paths, and so on, in a vicious cycle which creates extra work whose only function is to degrade precision [75] (fig. 3.6).

With 20 years of hindsight, we can now say that imprecision in kCFA happens because call strings are not a good abstraction of calling context. With a good abstraction, *a function usually behaves differently in different contexts*, so redundancy is minimized. With call strings, each program point potentially appears in a large number of contexts and the analysis results for many of them are the same [40]. Researchers have proposed BDDs as a way to tame redundancy [76, 74]. We believe it is better to use an abstraction that avoids introducing redundancy in the first place.

3.3.6 The root cause: call/return mismatch

A close look at the shortcomings of kCFA reveals that most of them are caused by call/return mismatch. Specifically,

- Call/return mismatch causes spurious data flows, which lower precision.
- Call/return mismatch causes spurious control flows, which hinder effective reasoning about stack change.
- Call/return mismatch causes syntactic brittleness.
- Imprecision induced by call/return mismatch creates extra work that slows down the analysis.

Only fake rebinding is not directly caused by call/return mismatch; it happens because kCFA does not solve the environment problem. In the next chapter, we show that CFA2 uses a single mechanism, a stack, for call/return matching and to avoid most fake rebinding.

Since kCFA, there has been a lot of subsequent work devoted to finding good abstractions of context (*e.g.*, [49, 75, 44, 43, 65]). These analyses provided insights on the different kinds of contexts and improved the state of the art of higher-order flow analysis. However, being finite-state, they all experience the limitations of call/return mismatch.

CHAPTER 4

The CFA2 analysis

This chapter presents CFA2, a higher-order flow analysis with unbounded call/return matching. The key insight is that instead of a finite-state machine, we can abstract a higher-order program to a pushdown automaton (or equivalent). By pushing return points on the stack, we always return to the right place. The name CFA2 stands for "a Context-Free Approach to Control-Flow Analysis."[1]

Like kCFA, CFA2 is an abstract interpretation of programs in CPS. We use a variant of CPS that forbids first-class control (sec. 4.1.3). First-class control presents special challenges for call/return matching, which we address in chapter 5.

The abstract semantics uses two environments for variable binding, a stack and a heap (sec. 4.2). Variable references are looked up in one or the other, depending on where they appear in the source code. Most references in typical programs are read from the stack, which results in significant precision gains. Also, CFA2 can filter certain bindings off the stack to sharpen precision (sec. 4.5).

Each abstract state has a stack of unbounded height. Hence, the abstract state space is infinite. To analyze it, we create an algorithm based on summarization, a dynamic-programming technique used by several pushdown-reachability analyses (sec. 4.3).

[1]The acronym is inspired by "ACL2: A Computational Logic for Applicative Common Lisp" [33]. We use "context-free" with its usual meaning from formal language theory, to indicate that CFA2 approximates valid executions as strings in a context-free language. Unfortunately, "context-free" means something else in program analysis. To avoid confusion, we use "monovariant" and "polyvariant" when we refer to the abstraction of calling context in program analysis. CFA2 is polyvariant (*aka* context sensitive), because it analyzes different calls to the same function in different environments.

4.1 Setting up the analysis

4.1.1 The Orbit stack policy

Our end goal is to obtain an abstraction of the concrete semantics that uses a stack to match calls and returns. To achieve this goal, we need to see how to manage the stack in a CPS setting, *i.e.*, when to push and pop.

The Orbit compiler [39, 38] compiles a CPS intermediate representation to final code that uses a stack. The abstract semantics of CFA2 follows Orbit's stack policy.[2] 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. This happens at $CCall$ call sites whose operator is a variable.
- 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.

4.1.2 Stack/heap split

The purpose of the stack in CFA2 is twofold: we use it for return point information and as an environment for variable binding. 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, the program

$(\lambda_1$`(x)` $(\lambda_2$`(y)` `(x (x y))))`

has a stack reference to `y` and two heap references to `x`.

Intuitively, only heap references may be captured in a heap-allocated closure. When we call a user function, we push a frame for its arguments, so

[2] Steele's Rabbit compiler for Scheme compiles CPS to final code that does not use a stack [67]. Steele's view is that argument evaluation pushes stack and function calls are GOTOs. Since arguments in CPS are not calls, argument evaluation is trivial and Rabbit never needs to push stack. By this approach, every call in CPS is a tail call. SML/NJ also uses this strategy [4]. CFA2 computes safe flow information, which can be used in compilers that follow Rabbit's stack policy. The workings of the abstract interpretation are independent of what style an implementor chooses for the final code.

we know that stack references are always bound in the top frame. We look up stack references in the top frame, and heap references in the heap. Stack lookups below the top frame never happen.

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 transformation 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 label" map $LL(\psi)$ returns the labels that are in the same set as ψ, *e.g.*, $LL(4) = \{2, 3, 4, 5\}$ and $LL(6) = \{1, 6\}$. The "label to variable" map $LV(\psi)$ returns all variables bound by any lambdas that belong in the same set as ψ, *e.g.*, $LV(4) = \{\texttt{y}, \texttt{k2}, \texttt{u}\}$ and $LV(6) = \{\texttt{x}, \texttt{k1}\}$. Notice that, for any ψ, $LV(\psi)$ contains exactly one continuation variable. Using LV, we give the following definition of stack and heap references.

Definition 7 (Stack and heap references)**.**

- Let ψ be a call site that refers to a variable x. The predicate $S_?(\psi, x)$ holds iff $x \in LV(\psi)$. We call x a **stack reference**.
- Let ψ be a call site that refers to a variable x. The predicate $H_?(\psi, x)$ holds iff $x \notin LV(\psi)$. We call x a **heap reference**.
- x is a **stack variable**, written $S_?(x)$, iff all its references satisfy $S_?$.
- x is a **heap variable**, written $H_?(x)$, 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 $\texttt{x} \notin \{\texttt{y}, \texttt{k2}, \texttt{u}\}$.

Continuations close over the stack, *e.g.*, the stack variable `k2` appears free in λ_3. Our stack policy must ensure that continuations can access the variables in their environment (see sec. 4.2).

$$
\begin{array}{rcl}
\hat{\varsigma} \in \widehat{UEval} & = & UCall \times Stack \times Heap \\
\hat{\varsigma} \in \widehat{UApply} & = & ULam \times \widehat{UProc} \times \widehat{CProc} \times Stack \times Heap \\
\hat{\varsigma} \in \widehat{CEval} & = & CCall \times Stack \times Heap \\
\hat{\varsigma} \in \widehat{CApply} & = & \widehat{CProc} \times \widehat{UProc} \times Stack \times Heap \\
\hat{d} \in \widehat{UProc} & = & Pow(ULam) \\
\hat{c} \in \widehat{CProc} & = & CLam + \{halt\} \\
fr, tf \in Frame & = & Var \rightharpoonup (\widehat{UProc} + \widehat{CProc}) \\
st \in Stack & = & Frame^* \\
h \in Heap & = & UVar \rightharpoonup \widehat{UProc}
\end{array}
$$

Figure 4.1: Domains

- When passing a continuation to a user function, we ensure that its environment is at the top of the stack.
- Before calling a continuation, we ensure that its environment is at the top of the stack.

4.1.3 Ruling out first-class control syntactically

Programs that use `call/cc` or similar constructs can perform actions that break call/return nesting, like jumping to a function that has already returned. First-class control complicates reasoning about the stack.

Without first-class control, functions only use their current continuation. This behavior is syntactically apparent in CPS; we can see it by observing the CPS translation of a few direct-style programs that do not use first-class control. In the example of the previous section, λ_2 only uses the continuation that is passed to it, which is `k2`. Thus, we can impose a simple syntactic constraint on CPS terms to rule out first-class control [15, 56]. We use the name CPS\1 for this variant of CPS.

Definition 8 (CPS\1)**.** A program is in CPS\1 iff the only continuation variable that can appear free in the body of a user lambda $(\lambda_l (u\ k)\ call)$ is k.

4.2 The semantics of CFA2

Like kCFA, the CFA2 semantics is an abstraction of the semantics of section 2.2.3. The abstract domains appear in fig. 4.1. An abstract user procedure

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

(a) Stack operations

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

$$
\hat{\mathcal{A}}_{\mathrm{k}}(q, st) \triangleq \begin{cases} q & Lam_?(q) \\ st(q) & Var_?(q) \end{cases}
$$

$$
\begin{aligned}
[\widehat{\mathrm{UEA}}]\ & (\llbracket (f\ e\ q)^l \rrbracket, st, h) \rightsquigarrow (ulam, \hat{\mathcal{A}}_{\mathrm{u}}(e, l, st, h), \hat{\mathcal{A}}_{\mathrm{k}}(q, st), st', h) \\
& ulam \in \hat{\mathcal{A}}_{\mathrm{u}}(f, l, st, h) \\
& st' = \begin{cases} pop(st) & Var_?(q) \\ st & Lam_?(q) \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
[\widehat{\mathrm{UAE}}]\ & (\llbracket (\lambda_l (u\ k)\ call) \rrbracket, \hat{d}, \hat{c}, st, h) \rightsquigarrow (call, st', h') \\
& st' = push([u \mapsto \hat{d},\ k \mapsto \hat{c}\,], st) \\
& h' = \begin{cases} h \sqcup [u \mapsto \hat{d}\,] & H_?(u) \\ h & S_?(u) \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
[\widehat{\mathrm{CEA}}]\ & (\llbracket (q\ e)^\gamma \rrbracket, st, h) \rightsquigarrow (\hat{\mathcal{A}}_{\mathrm{k}}(q, st), \hat{\mathcal{A}}_{\mathrm{u}}(e, \gamma, st, h), st', h) \\
& st' = \begin{cases} pop(st) & Var_?(q) \\ st & Lam_?(q) \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
[\widehat{\mathrm{CAE}}]\ & (\llbracket (\lambda_\gamma (u)\ call) \rrbracket, \hat{d}, st, h) \rightsquigarrow (call, st', h') \\
& st' = st[u \mapsto \hat{d}\,] \\
& h' = \begin{cases} h \sqcup [u \mapsto \hat{d}\,] & H_?(u) \\ h & S_?(u) \end{cases}
\end{aligned}
$$

(b) Semantics

Figure 4.2: Abstract semantics of CFA2

(member of the set $\widehat{UProc}$) is a set of user lambdas. An abstract continuation procedure (member of $\widehat{CProc}$) is either a continuation lambda or $halt$. A frame is a partial map from program variables to abstract values. A frame always maps user variables to user values and continuation variables to continuation values. A stack is a sequence of frames. All stack operations except $push$ are defined for non-empty stacks only (fig. 4.2a). A heap is a map from variables to abstract values. It contains only user bindings because, in CPS\1, every continuation variable is a stack variable.

Fig. 4.2b shows the transition rules. The initial state $\hat{\mathcal{I}}(pr)$ is a $\widehat{UApply}$ of the form $(pr, \{input\}, halt, \langle\rangle, \emptyset)$. We evaluate user terms using $\hat{\mathcal{A}}_{\mathrm{u}}$ and continuation terms using $\hat{\mathcal{A}}_{\mathrm{k}}$. Suppose a user variable u is referenced at call site ψ. We look up its value in the stack when $S_?(\psi, u)$ and in the heap otherwise. Note that even if u is a heap variable, we use the precise stack lookups for its stack references.

On transition from a $\widehat{UEval}$ state $\hat{\varsigma}$ to a $\widehat{UApply}$ state $\hat{\varsigma}'$ (rule $[\widehat{\mathrm{UEA}}]$), we first evaluate f, e and q. We non-deterministically choose one of the lambdas that flow to f as the operator in $\hat{\varsigma}'$. The change to the stack depends on q. If q is a variable, the call is a tail call so we pop the stack. If q is a lambda, it evaluates to a new closure whose environment is the top frame, hence we do not pop the stack.

In the $\widehat{UApply}$-to-$\widehat{Eval}$ transition (rule $[\widehat{\mathrm{UAE}}]$), we push a frame for the procedure's arguments. In addition, if u is a heap variable we must update its binding in the heap. The join operation $\sqcup$ is defined as:

$$(h \sqcup [u \mapsto \hat{d}\,])(x) \triangleq \begin{cases} h(x) & x \neq u \\ h(x) \cup \hat{d} & x = u \end{cases}$$

In a $\widehat{CEval}$-to-$\widehat{CApply}$ transition (rule $[\widehat{\mathrm{CEA}}]$), we are preparing for a call to a continuation closure, so we must bring its environment to the top of the stack. When q is a variable, the $\widehat{CEval}$ state is a function return and the continuation's environment is the second stack frame, so we pop. When q is a lambda, it is a newly created closure, so the stack does not change. Unlike $[\widehat{\mathrm{UEA}}]$, this transition is deterministic. Since we always know which continuation we are about to call, call/return mismatch *never* happens.

In the $\widehat{CApply}$-to-$\widehat{Eval}$ transition (rule $[\widehat{\mathrm{CAE}}]$), the top frame is the environment of $(\lambda_\gamma(u)\ call)$; 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.[3]

Examples When the analyzed program is not recursive, the stack size is bounded so we can enumerate all abstract states without diverging. Let's see how the analysis works on a simple program that applies the identity function twice and returns the result of the second call. The initial state $\hat{\mathcal{I}}(pr)$ is a $\widehat{UApply}$.

$$(\llbracket\texttt{(}\lambda\texttt{(i h)(i 1 (}\lambda_1\texttt{(n1)(i 2 (}\lambda_2\texttt{(n2)(h n2))))))}\rrbracket, \{\llbracket\texttt{(}\lambda_3\texttt{(x k)(k x))}\rrbracket\}, \mathit{halt}, \langle\rangle, \emptyset)$$

All variables in this example are stack variables, so the heap will be empty throughout the execution. In frames, we abbreviate lambda expressions by labeled lambdas. By rule $[\widehat{\mathrm{UAE}}]$, we push a frame for `i` and `h` and transition to a $\widehat{UEval}$ state.

$$(\llbracket\texttt{(i 1 (}\lambda_1\texttt{(n1)(i 2 (}\lambda_2\texttt{(n2)(h n2)))))}\rrbracket, \langle[\texttt{i} \mapsto \{\lambda_3\}, \texttt{h} \mapsto \mathit{halt}\,]\rangle, \emptyset)$$

We look up `i` in the top frame. Since the continuation argument is a lambda, we do not pop the stack. The next state is a $\widehat{UApply}$.

$$(\llbracket\texttt{(}\lambda_3\texttt{(x k)(k x))}\rrbracket, \{1\}, \lambda_1, \langle[\texttt{i} \mapsto \{\lambda_3\}, \texttt{h} \mapsto \mathit{halt}\,]\rangle, \emptyset)$$

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

$$(\llbracket\texttt{(k x)}\rrbracket, \langle[\texttt{x} \mapsto \{1\}, \texttt{k} \mapsto \lambda_1\,], [\texttt{i} \mapsto \{\lambda_3\}, \texttt{h} \mapsto \mathit{halt}\,]\rangle, \emptyset)$$

This is a $\widehat{CEval}$ state where the operator is a variable, so we pop a frame.

$$(\llbracket\texttt{(}\lambda_1\texttt{(n1)(i 2 (}\lambda_2\texttt{(n2)(h n2))))}\rrbracket, \{1\}, \langle[\texttt{i} \mapsto \{\lambda_3\}, \texttt{h} \mapsto \mathit{halt}\,]\rangle, \emptyset)$$

We extend the top frame to bind `n1` and jump to the body of λ_1.

$$(\llbracket\texttt{(i 2 (}\lambda_2\texttt{(n2)(h n2)))}\rrbracket, \langle[\texttt{n1} \mapsto \{1\}, \texttt{i} \mapsto \{\lambda_3\}, \texttt{h} \mapsto \mathit{halt}]\rangle, \emptyset)$$

The new call to `i` is also not a tail call, so we do not pop.

$$(\llbracket\texttt{(}\lambda_3\texttt{(x k)(k x))}\rrbracket, \{2\}, \lambda_2, \langle[\texttt{n1} \mapsto \{1\}, \texttt{i} \mapsto \{\lambda_3\}, \texttt{h} \mapsto \mathit{halt}]\rangle, \emptyset)$$

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

$$(\llbracket\texttt{(k x)}\rrbracket, \langle[\texttt{x} \mapsto \{2\}, \texttt{k} \mapsto \lambda_2\,], [\texttt{n1} \mapsto \{1\}, \texttt{i} \mapsto \{\lambda_3\}, \texttt{h} \mapsto \mathit{halt}]\rangle, \emptyset)$$

[3] All temporaries created by the CPS transformation are stack variables; but a compiler optimization may rewrite a program to create heap references to temporaries.

We pop a frame and jump to λ_2.

$$(\llbracket(\lambda_2\texttt{(n2)(h n2))}\rrbracket, \{2\}, \langle[\texttt{n1} \mapsto \{1\},\ \texttt{i} \mapsto \{\lambda_3\},\ \texttt{h} \mapsto \mathit{halt}]\rangle, \emptyset)$$

We extend the top frame to bind `n2` and jump to the body of λ_2.

$$(\llbracket\texttt{(h n2)}\rrbracket, \langle[\texttt{n2} \mapsto \{2\},\ \texttt{n1} \mapsto \{1\},\ \texttt{i} \mapsto \{\lambda_3\},\ \texttt{h} \mapsto \mathit{halt}]\rangle, \emptyset)$$

The operator is a variable, so we pop the stack. The next state is a final state, so the program terminates with value $\{2\}$.

$$(\mathit{halt}, \{2\}, \langle\rangle, \emptyset)$$

CFA2 is more resilient to η-expansion than kCFA (*cf.* sec. 3.3.3). If we η-expand λ_3 to `(`λ_3`(x k)((`λ_4`(y k2)(k2 y)) x k))`, CFA2 still finds the precise answer because the change did not create any heap references. Also, CFA2 is not affected by λ-binding the identity, unlike polymorphic splitting. However, if we change λ_3 to `(`λ_3`(x k)((`λ_4`(y k2)(k2 x)) x k))`, then both 1 and 2 flow to the heap reference to `x` and CFA2 will return $\{1, 2\}$.

4.2.1 Correctness

We show that the abstract semantics of CFA2 simulates the concrete semantics, using the methodology of section 2.1. Fig. 4.3 shows the abstraction function $|\cdot|_{\text{ca}}$, which maps concrete to abstract states. It is more involved than the one for kCFA because abstract states have a stack, so we must expose stack-related information hidden in ρ and ve of concrete states.

The abstraction of an Eval state ς of the form $(\mathit{call}, \rho, ve, t)$ is an $\widehat{\mathit{Eval}}$ state $\hat{\varsigma}$ with the same call site. We use the function $\mathit{toStack}$ to find the stack of $\hat{\varsigma}$. Let $\mathcal{L}(\mathit{call})$ be ψ and $iu_\lambda(\mathit{call})$ be λ_l. To reach ψ, control passed by 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 $\hat{\varsigma}$ (see rule $[\widehat{\text{CAE}}]$). Therefore, the domain of the top frame is a subset of $LV(l)$, *i.e.*, a subset of $LV(\psi)$. For each user variable $u_i \in (LV(\psi) \cap \text{dom}(\rho))$, the top frame contains $[u_i \mapsto |ve(u_i, \rho(u_i))|_{\text{ca}}]$. Let k be the sole continuation variable in $LV(\psi)$. If $ve(k, \rho(k))$ is halt (the return continuation is the top-level continuation), the rest of the stack is empty. If $ve(k, \rho(k))$ is (clam, ρ'), the second frame is for the user lambda in which (clam, ρ') 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.

$$|(call, \rho, ve, t)|_{\text{ca}} = (call,\ toStack(LV(\mathcal{L}(call)), \rho, ve),\ |ve|_{\text{ca}})$$

$$|(\langle ulam, \rho\rangle, d, c, ve, t)|_{\text{ca}} = (ulam, |d|_{\text{ca}}, |c|_{\text{ca}}, st, |ve|_{\text{ca}})$$

$$\text{where } st = \begin{cases} \langle\rangle & c = halt \\ toStack(LV(\mathcal{L}(clam)), \rho', ve) & c = (clam, \rho') \end{cases}$$

$$|(\langle clam, \rho\rangle, d, ve, t)|_{\text{ca}} = (clam, |d|_{\text{ca}},\ toStack(LV(\mathcal{L}(clam)), \rho, ve),\ |ve|_{\text{ca}})$$

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

$$|(ulam, \rho)|_{\text{ca}} = \{ulam\}$$

$$|(clam, \rho)|_{\text{ca}} = clam$$

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

$$|ve|_{\text{ca}} = \{\ (u, \bigcup_{(u,t)\in \text{dom}(ve)} |ve(u,t)|_{\text{ca}})\ :\ H_?(u)\}$$

$$toStack(\{u_1, \ldots, u_n, k\}, \rho, ve) \triangleq$$

$$\begin{cases} \langle[\overline{u_i \mapsto \hat{d}_i}][k \mapsto halt\,]\rangle & halt = ve(k, \rho(k)) \\ [\overline{u_i \mapsto \hat{d}_i}][k \mapsto clam] :: toStack(LV(\mathcal{L}(clam)), \rho', ve) & (clam, \rho') = ve(k, \rho(k)) \end{cases}$$

$$\text{where } \hat{d}_i = |ve(u_i, \rho(u_i))|_{\text{ca}}$$

Figure 4.3: From concrete to abstract states

The abstraction of a $UApply$ state over $\langle ulam, \rho\rangle$ is a $\widehat{UApply}$ state $\hat{\varsigma}$ whose operator is $ulam$. The stack of $\hat{\varsigma}$ is the stack in which the continuation argument was created, and we compute it using $toStack$ as above. Abstracting a $CApply$ is similar to the $UApply$ case, only now the top frame is the environment of the continuation operator.

The abstraction maps drop the time of the concrete states, since the abstract states do not use times. Unlike kCFA, the abstraction of a continuation closure is the corresponding lambda. When abstracting a variable environment ve, we only keep heap variables.

The ordering relation on abstract states (fig. 4.4) is similar to that of kCFA. Note that $st_1 \sqsubseteq st_2$ implies that st_1 and st_2 have the same length.

We can now state the simulation theorem for CFA2. The proof proceeds by case analysis on the concrete transition relation. It can be found in appendix A.

$$\begin{aligned}
\langle a_1, \ldots, a_n \rangle \sqsubseteq \langle b_1, \ldots, b_n \rangle \quad &\text{iff} \quad a_i \sqsubseteq b_i \quad \text{for } 1 \leqslant i \leqslant n \\
\hat{d}_1 \sqsubseteq \hat{d}_2 \quad &\text{iff} \quad \hat{d}_1 \subseteq \hat{d}_2 \\
h_1 \sqsubseteq h_2 \quad &\text{iff} \quad \forall x \in \mathrm{dom}(h_1).\ h_1(x) \sqsubseteq h_2(x) \\
tf_1 :: st_1 \sqsubseteq tf_2 :: st_2 \quad &\text{iff} \quad tf_1 \sqsubseteq tf_2 \ \wedge \ st_1 \sqsubseteq st_2 \\
tf_1 \sqsubseteq tf_2 \quad &\text{iff} \quad \forall x \in \mathrm{dom}(tf_1).\ tf_1(x) \sqsubseteq tf_2(x)
\end{aligned}$$

Figure 4.4: Ordering relation on abstract states

Theorem 9 (Simulation)**.** If $\varsigma \to \varsigma'$ and $|\varsigma|_{\mathrm{ca}} \sqsubseteq \hat{\varsigma}$, then there exists $\hat{\varsigma}'$ such that $\hat{\varsigma} \rightsquigarrow \hat{\varsigma}'$ and $|\varsigma'|_{\mathrm{ca}} \sqsubseteq \hat{\varsigma}'$.

4.2.2 The abstract semantics as a pushdown system

Let $\mathcal{RS}$ be the set of abstract states that are reachable from $\hat{\mathcal{I}}(pr)$. Since the size of the stack is not bounded, $\mathcal{RS}$ can be infinite. Thus, the abstract semantics of CFA2 does not correspond to a finite-state machine. CFA2 approximates higher-order programs as *pushdown systems*.

A pushdown system (PDS) is similar to a pushdown automaton, except it does not read input from a tape [21, 57]. Formally,

Definition (Pushdown System)**.** A pushdown system $\mathcal{P}$ is a triple (P, Γ, Δ). P is a finite set of control locations. Γ is a finite stack alphabet. A state of $\mathcal{P}$ is a pair of a control location $p \in P$ and a stack $w \in \Gamma^*$. Δ is a finite subset of $(P \times \Gamma) \times (P \times \Gamma^*)$. It contains rules of the form $(p, \gamma) \hookrightarrow (p', w)$, which define a transition relation $\Rrightarrow$ between states:

$$\text{If } (p, \gamma) \hookrightarrow (p', w) \text{ then } (p, \gamma w') \Rrightarrow (p', w w') \text{ for all } w' \in \Gamma^*$$

A transition may change the control location and replace the topmost stack symbol with a (possibly empty) string of stack symbols. The rest of the stack does not change and cannot influence the transition.

There is a natural connection between the CFA2 semantics and PDSs:

a) $Stack$ is the only infinite domain.
b) Frames below the top frame cannot influence a transition.

$\widehat{UEval}$ (non-tail call)	$(\llbracket(f\ e\ clam)^l\rrbracket, tf :: st, h) \rightsquigarrow (ulam, \hat{d}, \hat{c}, tf :: st, h)$
Push	$(\llbracket(f\ e\ clam)^l\rrbracket, h, tf) \hookrightarrow (\hat{d}, \hat{c}, h, \langle ulam, tf\rangle)$
$\widehat{UEval}$ (tail call)	$(\llbracket(f\ e\ k)^l\rrbracket, tf :: st, h) \rightsquigarrow (ulam, \hat{d}, \hat{c}, st, h)$
	$(\llbracket(f\ e\ k)^l\rrbracket, h, tf) \hookrightarrow (\hat{d}, \hat{c}, h, \langle ulam\rangle)$
$\widehat{CEval}$	$(\llbracket(clam\ e)^\gamma\rrbracket, tf :: st, h) \rightsquigarrow (clam, \hat{d}, tf :: st, h)$
	$(\llbracket(clam\ e)^\gamma\rrbracket, h, tf) \hookrightarrow (clam, \hat{d}, h, \langle tf\rangle)$
$\widehat{CEval}$ (function exit)	$(\llbracket(k\ e)^\gamma\rrbracket, tf :: st, h) \rightsquigarrow (clam, \hat{d}, st, h)$
Pop	$(\llbracket(k\ e)^\gamma\rrbracket, h, tf) \hookrightarrow (clam, \hat{d}, h, \langle\rangle)$
$\widehat{UApply}$	$(ulam, \hat{d}, \hat{c}, st, h) \rightsquigarrow (call, tf :: st, h')$
	$(\hat{d}, \hat{c}, h, ulam) \hookrightarrow (call, h', \langle tf\rangle)$
$\widehat{CApply}$	$(clam, \hat{d}, tf :: st, h) \rightsquigarrow (call, tf' :: st, h')$
	$(clam, \hat{d}, h, tf) \hookrightarrow (call, h', \langle tf'\rangle)$

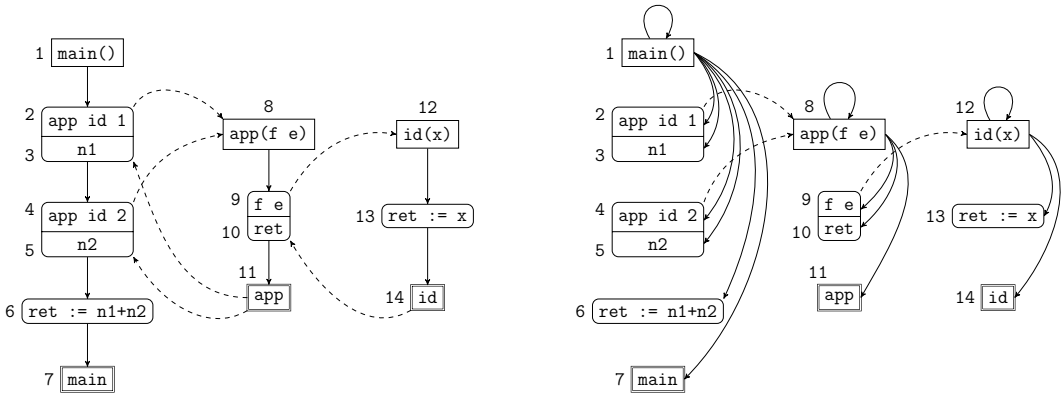
Figure 4.5: CFA2 as a pushdown system

We show the correspondence in fig. 4.5. Each abstract state $\hat{\varsigma}$ is a state (p, w) of a PDS $\mathcal{P}$ and each abstract transition gives rise to a rule in Δ. The stack st of $\hat{\varsigma}$ becomes w and the other components of $\hat{\varsigma}$ become p. (To illustrate this, we reorder the components in the PDS rules and put the stack last.) We make one tweak for $\widehat{UApply}$ states. $\hat{\mathcal{I}}(pr)$ is a $\widehat{UApply}$ and has an empty stack, but the left side of a PDS rule always has one stack symbol. Thus, we put the operator of a $\widehat{UApply}$ in the stack to match the PDS-rule format.

The set of reachable states $\mathcal{RS}$ of a PDS $\mathcal{P}$ may be infinite because Γ^* is infinite. Therefore, we cannot enumerate all states in $\mathcal{RS}$. However, $\mathcal{RS}$ is regular [21]. Pushdown-reachability algorithms explore $\mathcal{RS}$ by using a dynamic-programming technique called *summarization*, which we describe in the next section.

4.3 Exploring the infinite state space

We use summarization to explore the state space in CFA2. Our algorithm is based on Sharir and Pnueli's functional approach [59, pg. 207], adapted to the terminology of Reps *et al.* [52]. These algorithms require that we know all call sites of a function. Therefore, they do not apply directly to



(a) Control-flow graph (b) Path and call edges

Figure 4.6: Summarization for `app/id`

higher-order languages, because we cannot find all call sites of a function by looking at a program's source code. We need a *search-based* variant of summarization, which records callers as it discovers them.

4.3.1 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 f. We start computing inside the procedure. While doing so, we are visiting several program points inside f 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 of f 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 stack s we can reach n' with stack $s's$.

Let's return to the `app/id` example and use summarization to find which nodes of the graph are reachable from node 1. Fig. 4.6a shows the control-flow graph. We find reachable nodes by recording *path edges*, *i.e.*, edges whose source is the entry of a procedure and whose target is some reachable 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 $\langle 4, 8\rangle$ 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 $\langle 1, 7\rangle$. Fig. 4.6b shows all path and call edges.

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 the stack is not an explicit part of the model, 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 4.3.3. The next section describes the local semantics, which we use in the algorithm for transitions that do not shrink the stack.

4.3.2 Local semantics

Summarization-based algorithms operate 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* and define a map $|\cdot|_{\text{al}}$ from abstract to local states (fig. 4.7). Intuitively, a local state is like an abstract state but with a single frame instead of a stack. Discarding the rest of the stack makes the local state space finite; keeping the top frame allows precise lookups for stack references.

The local semantics describes executions that do not touch the rest of the stack, in other words, executions where functions do not return. Thus, a $\widetilde{CEval}$ state with call site $(k\ e)^\gamma$ has no successor in this semantics. Since functions do not call their continuations, the frames of local states contain only user bindings. Local steps are otherwise similar to abstract steps. The metavariable $\tilde{\varsigma}$ ranges over local states. We define the map $|\cdot|_{\text{cl}}$ from concrete

$$
\begin{array}{rcl}
\widetilde{Eval} & = & Call \times \widetilde{Stack} \times Heap \\
\widetilde{UApply} & = & ULam \times \widehat{UProc} \times Heap \\
\widetilde{CApply} & = & \widehat{CProc} \times \widehat{UProc} \times \widetilde{Stack} \times Heap \\
\widetilde{Frame} & = & UVar \rightharpoonup \widehat{UProc} \\
\widetilde{Stack} & = & \widetilde{Frame}
\end{array}
$$

(a) Domains

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

(b) From abstract to local states

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

$$
\begin{array}{ll}
[\widetilde{\text{UEA}}] & (\llbracket (f\ e\ q)^l \rrbracket, tf, h) \twoheadrightarrow (ulam, \tilde{\mathcal{A}}_{\text{u}}(e, l, tf, h), h) \\
& ulam \in \tilde{\mathcal{A}}_{\text{u}}(f, l, tf, h) \\
[\widetilde{\text{UAE}}] & (\llbracket (\lambda_l (u\ k)\ call) \rrbracket, \hat{d}, h) \twoheadrightarrow (call, [u \mapsto \hat{d}], h') \\
& h' = \begin{cases} h \sqcup [u \mapsto \hat{d}] & H_?(u) \\ h & S_?(u) \end{cases} \\
[\widetilde{\text{CEA}}] & (\llbracket (clam\ e)^\gamma \rrbracket, tf, h) \twoheadrightarrow (clam, \tilde{\mathcal{A}}_{\text{u}}(e, \gamma, tf, h), tf, h) \\
[\widetilde{\text{CAE}}] & (\llbracket (\lambda_\gamma (u)\ call) \rrbracket, \hat{d}, tf, h) \twoheadrightarrow (call, tf', h') \\
& tf' = tf[u \mapsto \hat{d}] \\
& h' = \begin{cases} h \sqcup [u \mapsto \hat{d}] & H_?(u) \\ h & S_?(u) \end{cases}
\end{array}
$$

(c) Semantics

Figure 4.7: Local semantics of CFA2

to local states to be $|\cdot|_{\mathrm{al}} \circ |\cdot|_{\mathrm{ca}}$.

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 10 (Classification of states)**.**

- A $UApply$ state is an **Entry**—control is at the beginning of a function.
- A $CEval$ state where the operator is a variable is an **Exit-Ret**—a function is about to return a result to its context.
- 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-Ret, 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.

The next example shows that tail calls require generalizing the notion of summary edges. Consider the `app/id` program written in CPS.

```
((λ (app id k)
    (app id 1 (λ1(n1) (app id 2 (λ2(n2) (+ n1 n2 k))))))
 (λ (f e k) (f e k))
 (λ (x k) (k x))
 halt)
```

The call `(f e k)` in the body of `app` is a tail call, so no continuation is born there. Upon return from the first call to `id`, we must be careful to pass the result to λ_1. Also, we must restore the environment of the first call to `app`, *not* the environment of the tail call. Similarly, the second call to `id` must return to λ_2 and restore the correct environment. We achieve these with a "cross-procedure" summary from the entry of `app` to call site `(k x)`, which is the exit of `id`. This transitive nature of summaries is essential for tail calls.

4.3.3 Workset algorithm

The algorithm for CFA2 is shown in fig. 4.8. It is a search-based summarization for higher-order programs with tail calls. Its goal is to compute which local states are reachable from the initial state of a program through paths that respect call/return matching.

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 an Exit-Ret, not necessarily in the same procedure (due to tail calls). Summaries carry an important message: *each continuation 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 $Summary$. $Final$ 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 $\tilde{\mathcal{I}}(pr)$ is defined as $|\mathcal{I}(pr)|_{\text{cl}}$. The helper function $succ(\tilde{\varsigma})$ returns the successor(s) of $\tilde{\varsigma}$ according to the local semantics.

Edge processing Each edge $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ is processed in one of four ways, depending on $\tilde{\varsigma}_2$. If $\tilde{\varsigma}_2$ is an entry, a return or an inner state (line 6), 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 27).

If $\tilde{\varsigma}_2$ is a call (line 8) 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 10). 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 12). The function `Update` is responsible for computing the return

```
1   Summary, Callers, TCallers, Final ← ∅
2   Seen, W ← {(Ĩ(pr), Ĩ(pr))}
3   while W ≠ ∅
4     remove (ς̃1, ς̃2) from W
5     switch ς̃2
6       case ς̃2 of Entry, CApply, Inner-CEval
7         for the ς̃3 in succ(ς̃2), Propagate(ς̃1, ς̃3)
8       case ς̃2 of Call
9         foreach ς̃3 in succ(ς̃2)
10          Propagate(ς̃3, ς̃3)
11          insert (ς̃1, ς̃2, ς̃3) in Callers
12          foreach (ς̃3, ς̃4) in Summary, Update(ς̃1, ς̃2, ς̃4)
13      case ς̃2 of Exit-Ret
14        if ς̃1 = Ĩ(pr) then
15          Final(ς̃2)
16        else
17          insert (ς̃1, ς̃2) in Summary
18          foreach (ς̃3, ς̃4, ς̃1) in Callers, Update(ς̃3, ς̃4, ς̃2)
19          foreach (ς̃3, ς̃4, ς̃1) in TCallers, Propagate(ς̃3, ς̃2)
20      case ς̃2 of Exit-TC
21        foreach ς̃3 in succ(ς̃2)
22          Propagate(ς̃3, ς̃3)
23          insert (ς̃1, ς̃2, ς̃3) in TCallers
24          foreach (ς̃3, ς̃4) in Summary, Propagate(ς̃1, ς̃4)
25
26  Propagate(ς̃1, ς̃2) ≜
27    if (ς̃1, ς̃2) ∉ Seen then insert (ς̃1, ς̃2) in Seen and W
28
29  Update(ς̃1, ς̃2, ς̃3) ≜
30    ς̃2 of the form (⟦(f e clam)^l⟧, tf2, h2)
31    ς̃3 of the form (⟦(k e3)^γ⟧, tf3, h3)
32    d̂ ← Ã_u(e3, γ, tf3, h3)
33    ς̃ ← (clam, d̂, tf2, h3)
34    Propagate(ς̃1, ς̃)
35
36  Final(ς̃) ≜
37    ς̃ of the form (⟦(k e)^γ⟧, tf, h)
38    insert (halt, Ã_u(e, γ, tf, h), ∅, h) in Final
```

Figure 4.8: Workset algorithm

state. We find the return value $\hat{d}$ by evaluating the expression e_3 passed to the continuation (lines 31 – 32). Since we are returning to $clam$, we must restore the environment of its creation which is tf_2. The new state $\tilde{\varsigma}$ is the corresponding return of $\tilde{\varsigma}_2$, so we propagate $(\tilde{\varsigma}_1, \tilde{\varsigma})$ (lines 33 – 34).

If $\tilde{\varsigma}_2$ is an Exit-Ret and $\tilde{\varsigma}_1$ is the initial state (lines 14 – 15), then $\tilde{\varsigma}_2$'s successor is a final state (lines 37 – 38). 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 17 – 18). Last, consider the case of a tail call $\tilde{\varsigma}_4$ to $\tilde{\varsigma}_1$ (line 19). 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 $\tilde{\varsigma}_2$ (line 19).

If $\tilde{\varsigma}_2$ is a tail call (line 20), we find its successors and record the call in $TCallers$ (lines 21 – 23). If a successor of $\tilde{\varsigma}_2$ goes to an exit, we propagate a cross-procedure summary transitively (line 24). Table 4.1 shows a complete run of the algorithm for a small program.

4.3.4 Correctness

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 terminates.

We obviously cannot visit an infinite number of abstract states. To establish soundness, we show that for each state $\hat{\varsigma} \in \mathcal{RS}$, the algorithm visits $|\hat{\varsigma}|_{\text{al}}$ (theorem 13). In essence, the algorithm visits the reachable *control states* of a PDS (see sec. 4.2.2).

Does it also visit other states, which result in spurious flows that do not happen in the abstract semantics? For example, a sound but useless algorithm would add all pairs of local states in $Seen$. We establish the completeness of CFA2 by proving that every visited edge corresponds to an abstract flow (theorem 14), which means that there is no loss in precision when going from abstract to local states.

The theorems use two definitions. The first associates a state $\hat{\varsigma}$ with its *corresponding entry*, *i.e.*, the entry of the procedure that contains $\hat{\varsigma}$. The second finds all entries that reach $CE_p(\hat{\varsigma})$ through tail calls. We include the

Name	Kind	Value
$\tilde{\mathcal{I}}(pr)$	Entry	$(\llbracket(\lambda_2\texttt{(id h)(id 1 }(\lambda_3\texttt{(u)(id 2 h))))}\rrbracket, \{\llbracket(\lambda_1\texttt{(x k)(k x))}\rrbracket\}, \emptyset)$
$\tilde{\varsigma}_1$	Call	$(\llbracket\texttt{(id 1 }(\lambda_3\texttt{(u)(id 2 h)))}\rrbracket, [\texttt{id} \mapsto \{\lambda_1\}], \emptyset)$
$\tilde{\varsigma}_2$	Entry	$(\lambda_1, \{1\}, \emptyset)$
$\tilde{\varsigma}_3$	Exit-Ret	$(\llbracket\texttt{(k x)}\rrbracket, [\texttt{x} \mapsto \{1\}], \emptyset)$
$\tilde{\varsigma}_4$	$\widetilde{CApply}$	$(\lambda_3, \{1\}, [\texttt{id} \mapsto \{\lambda_1\}], \emptyset)$
$\tilde{\varsigma}_5$	Exit-TC	$(\llbracket\texttt{(id 2 h)}\rrbracket, [\texttt{id} \mapsto \{\lambda_1\}, \texttt{u} \mapsto \{1\}], \emptyset)$
$\tilde{\varsigma}_6$	Entry	$(\lambda_1, \{2\}, \emptyset)$
$\tilde{\varsigma}_7$	Exit-Ret	$(\llbracket\texttt{(k x)}\rrbracket, [\texttt{x} \mapsto \{2\}], \emptyset)$
$\tilde{\varsigma}_8$	$\widetilde{CApply}$	$(halt, \{2\}, \emptyset, \emptyset)$

W	*Summary*	*Callers*	*TCallers*	*Final*
$(\tilde{\mathcal{I}}(pr), \tilde{\mathcal{I}}(pr))$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_1)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$(\tilde{\varsigma}_2, \tilde{\varsigma}_2)$	$\emptyset$	$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_1, \tilde{\varsigma}_2)$	$\emptyset$	$\emptyset$
$(\tilde{\varsigma}_2, \tilde{\varsigma}_3)$	$\emptyset$	$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_1, \tilde{\varsigma}_2)$	$\emptyset$	$\emptyset$
$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_4)$	$(\tilde{\varsigma}_2, \tilde{\varsigma}_3)$	$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_1, \tilde{\varsigma}_2)$	$\emptyset$	$\emptyset$
$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_5)$	$(\tilde{\varsigma}_2, \tilde{\varsigma}_3)$	$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_1, \tilde{\varsigma}_2)$	$\emptyset$	$\emptyset$
$(\tilde{\varsigma}_6, \tilde{\varsigma}_6)$	$(\tilde{\varsigma}_2, \tilde{\varsigma}_3)$	$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_1, \tilde{\varsigma}_2)$	$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_5, \tilde{\varsigma}_6)$	$\emptyset$
$(\tilde{\varsigma}_6, \tilde{\varsigma}_7)$	$(\tilde{\varsigma}_2, \tilde{\varsigma}_3)$	$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_1, \tilde{\varsigma}_2)$	$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_5, \tilde{\varsigma}_6)$	$\emptyset$
$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_7)$	$(\tilde{\varsigma}_2, \tilde{\varsigma}_3),\ (\tilde{\varsigma}_6, \tilde{\varsigma}_7)$	$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_1, \tilde{\varsigma}_2)$	$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_5, \tilde{\varsigma}_6)$	$\emptyset$
$\emptyset$	$(\tilde{\varsigma}_2, \tilde{\varsigma}_3),\ (\tilde{\varsigma}_6, \tilde{\varsigma}_7)$	$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_1, \tilde{\varsigma}_2)$	$(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_5, \tilde{\varsigma}_6)$	$\tilde{\varsigma}_8$

Table 4.1: A complete run of CFA2. Note that λ_1 is applied twice and returns to the correct context both times. The program evaluates to 2. For brevity, we first show all reachable states and then refer to them by their names. $\tilde{\mathcal{I}}(pr)$ shows the whole program; in the other states we abbreviate lambdas by their labels. All heaps are $\emptyset$ because there are no heap variables. The rows of the table show the contents of the sets at line 3 for each iteration. *Seen* contains all pairs entered in W.

proofs of the theorems in appendix A.

Definition 11. The Corresponding Entry $CE_p(\hat{\varsigma})$ of a state $\hat{\varsigma}$ in a path p is:

- $\hat{\varsigma}$, if $\hat{\varsigma}$ is an Entry
- $\hat{\varsigma}_1$, if $\hat{\varsigma}$ is not an Entry, $p = p_1 \rightsquigarrow \hat{\varsigma}_1 \rightsquigarrow^* \hat{\varsigma}_2 \rightsquigarrow \hat{\varsigma} \rightsquigarrow p_2$, $\hat{\varsigma}_2$ is not an Exit-Ret, and $CE_p(\hat{\varsigma}_2) = \hat{\varsigma}_1$
- $\hat{\varsigma}_1$, if $\hat{\varsigma}$ is not an Entry, $p = p_1 \rightsquigarrow \hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}_2 \rightsquigarrow \hat{\varsigma}_3 \rightsquigarrow^+ \hat{\varsigma}_4 \rightsquigarrow \hat{\varsigma} \rightsquigarrow p_2$, $\hat{\varsigma}_2$ is a Call and $\hat{\varsigma}_4$ is an Exit-Ret, $CE_p(\hat{\varsigma}_2) = \hat{\varsigma}_1$, and $\hat{\varsigma}_3 \in CE^*_p(\hat{\varsigma}_4)$

Definition 12. For state $\hat{\varsigma}$ and path p, $CE^*_p(\hat{\varsigma})$ is the smallest set such that:

- $CE_p(\hat{\varsigma}) \in CE^*_p(\hat{\varsigma})$
- $CE^*_p(\hat{\varsigma}_1) \subseteq CE^*_p(\hat{\varsigma})$, when $p = p_1 \rightsquigarrow \hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^* \hat{\varsigma} \rightsquigarrow p_2$, $\hat{\varsigma}_1$ is a Tail Call, $\hat{\varsigma}_2$ is an Entry, and $\hat{\varsigma}_2 = CE_p(\hat{\varsigma})$

Theorem 13 (Soundness)**.** Let $p = \hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}$. Then, after summarization:

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

Theorem 14 (Completeness)**.** After summarization:

- For each $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ in Seen, there exist states $\hat{\varsigma}_1, \hat{\varsigma}_2 \in \mathcal{RS}$ such that $\hat{\varsigma}_1 \rightsquigarrow^* \hat{\varsigma}_2$ and $\tilde{\varsigma}_1 = |\hat{\varsigma}_1|_{\text{al}}$ and $\tilde{\varsigma}_2 = |\hat{\varsigma}_2|_{\text{al}}$ and $\hat{\varsigma}_1 \in CE^*_p(\hat{\varsigma}_2)$
- For each $\tilde{\varsigma}$ in Final, there exists a final state $\hat{\varsigma} \in \mathcal{RS}$ such that $\tilde{\varsigma} = |\hat{\varsigma}|_{\text{al}}$

4.4 Without heap variables, CFA2 is exact

Joining of abstract values introduces imprecision in a flow analysis. In CFA2, we join only when we insert a value in the heap. Therefore, CFA2 is exact for programs without heap variables.

We refer to the set of programs with stack variables only as the *stack λ-calculus*, λ_S. Note that λ_S includes non-terminating programs such as

```
((λ (x) (x x)) (λ (x) (x x)))
```

The workset algorithm finds in finite time that this program does not terminate; halt is never applied, so when the algorithm ends, Final is empty. Let T_1 be the term `(λ (x) (x x))`. The evaluation of $(T_1\ T_1)$ has a finite number of states because the iteration happens in tail position.

$$(T_1\ T_1) \to (T_1\ T_1) \to (T_1\ T_1) \to \ldots$$

The stack λ-calculus also includes programs whose evaluation has infinitely many states. Let T_2 be the term `(λ (x) (x (x x)))`. The call-by-value evaluation of $(T_2\ T_2)$ is

$$(T_2\ T_2) \to (T_2\ (T_2\ T_2)) \to (T_2\ (T_2\ (T_2\ T_2))) \to \ldots$$

CFA2 still finds that this program does not terminate.

Using the CFA2 workset algorithm, every λ_S program can be converted to an equivalent PDS $\mathcal{P}$. Since there is no joining of abstract values, every $\widetilde{UEval}$ state has a single successor, so $\mathcal{P}$ is deterministic. It is an open question whether every deterministic PDS can be converted to an equivalent λ_S program.

For the ordinary λ-calculus with single-argument functions, λ_S is rather restrictive. However, in the λ-calculus with multi-argument functions, λ_S contains interesting terms such as Church booleans and boolean connectives, *e.g.*, TRUE is `(λ (x y) x)`, AND is `(λ (x y) (x y x))`.[4] Naturally, the Y combinator is not in λ_S.

```
(λ (f) ((λ (x) (f (λ (y) ((x x) y))))
        (λ (x) (f (λ (y) ((x x) y))))))
```

Y recurs by passing around closures over `(λ (x) (f (λ (y) ((x x) y))))`. Heap references are essential for unrestricted computation.

4.5 Stack filtering

When a flow analysis cannot see that two references of a variable are bound in the same environment, fake rebinding may occur (sec. 3.3.4). In CFA2, we can use the stack/heap distinction to prevent fake rebinding for a large class of variable references.

All stack references of a variable are bound in the same environment, the top stack frame, so they must all be bound to the same value. We can adapt rule $[\widehat{\text{UEA}}]$ in the abstract semantics to achieve that.

[4]Here, we mean the generalized λ-calculus where a multi-argument function cannot be partially applied, so it is not just syntactic sugar for the curried version in the ordinary λ-calculus.

$$
\begin{array}{l}
(\llbracket (f\ e\ q)^l \rrbracket, st, h) \rightsquigarrow (ulam, \hat{\mathcal{A}}_{\mathrm{u}}(e, l, st, h), \hat{\mathcal{A}}_{\mathrm{k}}(q, st), st', h) \\
ulam \in \hat{\mathcal{A}}_{\mathrm{u}}(f, l, st, h) \\
st' = \begin{cases} pop(st) & Var_?(q) \\ st & Lam_?(q) \wedge (H_?(l, f) \vee Lam_?(f)) \\ st[f \mapsto \{ulam\}] & Lam_?(q) \wedge S_?(l, f) \end{cases}
\end{array}
$$

Let's say control reaches a user call site $(f\ e\ q)^l$ whose operator is a stack reference. We pick a lambda $ulam$ and call it. If control reaches other stack references of f after we return from $ulam$, they must be bound to $ulam$, otherwise there is a spurious flow. Thus, we commit to $ulam$ by removing all other values from the set $st(f)$.

We cannot change rule $\widetilde{[\mathrm{UEA}]}$ of the local semantics because $\widetilde{UApply}$ states do not have stacks. Instead, we filter when a callee returns to its caller. We add the entry of the callee as an extra argument to `Update`, before the Exit-Ret. The call at line 12 becomes `Update(`$\tilde{\varsigma}_1$`,`$\tilde{\varsigma}_2$`,`$\tilde{\varsigma}_3$`,`$\tilde{\varsigma}_4$`)` and the call at line 18 becomes `Update(`$\tilde{\varsigma}_3$`,`$\tilde{\varsigma}_4$`,`$\tilde{\varsigma}_1$`,`$\tilde{\varsigma}_2$`)`. The code of `Update` is

$$
\begin{array}{l}
\texttt{Update}(\tilde{\varsigma}_1,\ \tilde{\varsigma}_2,\ \tilde{\varsigma}_3,\ \tilde{\varsigma}_4) \triangleq \\
\quad \tilde{\varsigma}_2\ \texttt{of the form}\ (\llbracket (f\ e\ clam)^l \rrbracket,\ tf_2,\ h_2) \\
\quad \tilde{\varsigma}_4\ \texttt{of the form}\ (\llbracket (k\ e_4)^\gamma \rrbracket,\ tf_4,\ h_4) \\
\quad \hat{d} \leftarrow \tilde{\mathcal{A}}_{\mathrm{u}}(e_4,\ \gamma,\ tf_4,\ h_4) \\
\quad tf \leftarrow \begin{cases} tf_2[f \mapsto \{\pi_1(\tilde{\varsigma}_3)\}] & S_?(l, f) \\ tf_2 & H_?(l, f) \vee Lam_?(f) \end{cases} \\
\quad \tilde{\varsigma} \leftarrow (clam,\ \hat{d},\ tf,\ h_4) \\
\quad \texttt{Propagate}(\tilde{\varsigma}_1, \tilde{\varsigma})
\end{array}
$$

We check if the operator f at the call site is a stack reference. If so, we filter the frame of the return state to keep only the lambda that was called.

4.6 Complexity

A simple calculation shows that the CFA2 workset algorithm is in EXPTIME. The size of the domain of $Heap$ is n and the size of the range is 2^n, so there are 2^{n^2} heaps. Since local states contain heaps, there are exponentially many local states. Thus, $Seen$ contains exponentially many edges. The running time of the algorithm is bounded by the number of edges in W times the cost of each iteration. W contains edges from $Seen$ only, so its size is exponential

```
(let* ((merger   (λ1(f) (λ2(_) f)))
       (_        (merger (λ3(x) x)))
       (twolams  ((merger (λ4(y) y)) _))
       (f1       twolams)
       (_        (f1 _)^1)
       (f2       twolams)
       (_        (f2 _)^2)
                 ⋮
       (fn       twolams)
       (_        (fn _)^n))
  _ )
```

Figure 4.9

in n. Therefore, the algorithm requires exponential time. In appendix B, we show that the algorithm runs in time $O(n^8 2^{9n^2+3n})$.

What kinds of programs trigger the exponential behavior? One possibility is to create exponentially many frames by exploiting the stack-filtering mechanism. Consider the program of fig. 4.9, suggested to us by Danny Dubé. The code is in direct style for brevity; the let-bound variables would be bound by continuation lambdas in the equivalent CPS program. The only heap reference appears in the body of λ_2. We use underscores for unimportant expressions.

The `merger` takes a function, binds `f` to it and returns a closure that ignores its argument and returns `f`. We call the `merger` twice so that `f` is bound to $\{\lambda_3, \lambda_4\}$ in the heap. After the second call, `twolams` is bound to $\{\lambda_3, \lambda_4\}$ in the top frame. We bind `f1` to $\{\lambda_3, \lambda_4\}$. At call site 1, execution splits in two branches. One calls λ_3 and filters the binding of `f1` in the top frame to $\{\lambda_3\}$. The other calls λ_4 and filters the binding to $\{\lambda_4\}$. Each branch will split into two more branches at call 2, *etc*. By binding each `fi` to a set of two elements and applying it immediately, we force a strong update and create exponentially many frames.

Even in the semantics without stack filtering, we can create an exponential number of frames (fig. 4.10). This time, we take advantage of the fact that $\widehat{UEval}$ states can have many successors that are analyzed independently. Again, we bind `twolams` to a set of two lambdas: λ_3 ignores its argument and returns `"foo"` and λ_4 ignores its argument and returns `"bar"`. Before the call

```
(let* ((merger   (λ1(f) (λ2(_) f)))
       (_        (merger (λ3(_) "foo")))
       (twolams  ((merger (λ4(_) "bar")) _))
       (x1       (twolams _)^1)
       (x2       (twolams _)^2)
                 ⋮
       (xn       (twolams _)^n))
  _ )
```

Figure 4.10

at call site 1, the variables `x1`, ..., `xn` are not bound in the top frame. The call has two successor $\widehat{UApply}$ states, for λ_3 and λ_4. So, one branch of the execution will bind `x1` to `"foo"` in the top frame and the other one to `"bar"`. At call 2, each branch will split in two more branches, *etc*. The analysis will branch 2^n times and each path will bind `x1`, ..., `xn` to a different combination of `"foo"` and `"bar"`.

This example shows that CFA2 is in the category of *path-sensitive* analyses, because it uses different environments on different execution paths. As an aside, if we run 0CFA on this program using a variable environment *per state*, instead of a single global variable environment, it also needs exponential time (in contrast to 0CFA with a single global environment, which is a cubic analysis).

4.6.1 Towards a PTIME algorithm

We tried to keep the algorithm of fig. 4.8 simple because it is meant to be a model. Several widening techniques can speed up convergence at the cost of lowering precision.

Instead of having one heap per state, we can use Shivers's timestamp technique [61, ch. 5]. We maintain a global heap and a global counter. Every time the heap changes, we increase the counter. The heap component of each state is now a timestamp, which shows the value of the counter when the state was created. Thus, comparing the heap components of states takes constant time. $Heap$ is a lattice of height $O(n^2)$. Since the global heap grows monotonically, it can change at most $O(n^2)$ times during the analysis.

In the semantics without stack filtering, we can bound the number of

frames by widening. The program of fig. 4.10 creates an exponential number of states for the program point `(xn (twolams _)`^{n}`)`, each with a different frame. If instead we join the results after every call to `twolams`, we have a single frame $[\texttt{x1} \mapsto \{\lambda_3, \lambda_4\}, \ldots, \texttt{xn} \mapsto \{\lambda_3, \lambda_4\}]$. For example, assume that we want to propagate an edge $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$, where $\tilde{\varsigma}_2$ is a $\widetilde{UEval}$ of the form $([\![(f\ e\ q)^l]\!], tf, h)$. We search $Seen$ for a pair $(\tilde{\varsigma}_1, ([\![(f\ e\ q)^l]\!], tf', h'))$. (Note that tf' and h' can be $\bot$ if it is the first time that we reach $(f\ e\ q)^l$ from $\tilde{\varsigma}_1$.) Then, we propagate $(\tilde{\varsigma}_1, ([\![(f\ e\ q)^l]\!], tf \sqcup tf', h \sqcup h'))$. This way, $\tilde{\varsigma}_2$ grows monotonically. Also, $\widetilde{Stack}$ is a lattice of height $O(n^2)$. Therefore, $\tilde{\varsigma}_1$ can reach at most $O(n^4)$ states over call site $(f\ e\ q)^l$.

By widening the target states of path edges, we ensure that each entry can reach a polynomial number of states. Thus, to make the state space (and the running time) polynomial, it suffices to bound the number of entries by a polynomial, which can also be achieved by widening.

CHAPTER 5

CFA2 for first-class control

Algorithms based on summarization, including the one in chapter 4, 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* (*e.g.*, in Python and JavaScript) are functions 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* (*e.g.*, in Simula67 [13] and Lua) can also suspend and resume their execution, but are more expressive than generators because they can specify where to pass control when they yield. *First-class continuations* reify the rest of the computation as a value. Undelimited continuations (`call/cc` in Scheme and SML/NJ [4]) capture the entire stack. Delimited continuations [20, 14], found in Scala [54] and some Schemes, capture part of the stack. Continuations can express generators and coroutines, as well as multi-threading [71, 63] and Prolog-style backtracking.

All these operators provide a rich variety of control behaviors. In this chapter, we generalize CFA2 to languages with such operators. We first define a variant of CPS that permits first-class control, unlike CPS\1 (sec. 5.1). Then, we present the abstract semantics for this new variant (sec. 5.2). The semantics detects continuations that may escape and copies the stack into the heap. For brevity, the semantics does not use stack filtering; it is easy to add using the ideas of sec. 4.5. In section 5.3, we generalize summarization to handle escaping continuations with a new kind of summary edge. Section 5.4 shows how to increase precision for continuations that are only used downward.

Note For a language with exceptions but no first-class control, the material in this chapter is not necessary. In chapter 6, we show that exceptions can be modeled precisely by incorporating them in summaries.

5.1 Restricted CPS

To handle first-class control in CFA2, we must allow capture of continuation variables in user closures. Thus, we must abandon CPS\1. However, we do not need general CPS. We propose a variant which is restrictive enough to permit effective reasoning about calls and returns, but permissive enough to express the same control operators as general CPS. We call this variant Restricted CPS (RCPS).

Definition 15 (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: *after a continuation escapes, it can only be called*; it cannot be passed as an argument again. For example, the CPS-translation of `call/cc`,

```
(λ (f cc) (f (λ (v k) (cc v)) cc))
```

is a valid RCPS term. Terms such as

```
(λ (x k) (k (λ (y k2) (y 123 k))))
```

are not valid. We can easily transform the previous term (and any CPS term) to a valid RCPS term by η-expanding to bring the free reference into operator position:

```
(λ (x k) (k (λ (y k2) (y 123 (λ (u) (k u))))))
```

These two very similar terms highlight the effect of RCPS on stack behavior. In the first case, 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, a new continuation is born at call site `(y 123 (λ (u) (k u)))`, so no stack change is required. Thus, RCPS forces all exotic stack change to happen when calling an escaping continuation, not in user call sites.

5.2 Abstract semantics

This section describes how to change the CFA2 semantics to make it work for RCPS. Remember that continuations close over the stack (sec. 4.1.2). Hence, when a continuation $\hat{c}$ escapes, we need to save the stack and restore it when $\hat{c}$ gets called. We do that by copying the stack into the heap.

Fig. 5.1a shows the domains. They are the same as before (see fig. 4.1), except that the heap can also contain continuation bindings. Fig. 5.1b shows the transition rules. First-class control shows up in rules $[\widehat{\text{UAE}}]$ and $[\widehat{\text{CEA}}]$; the other two rules are unchanged.

Upon entering the body of a user function (rule $[\widehat{\text{UAE}}]$), we push a new frame that binds the formals to the arguments, as before. If k is a heap variable, the continuation that flows to k may escape and later get called. Thus, we copy the continuation closure into the heap (the code point $\hat{c}$ but also the environment st).

Before calling a continuation, we must restore its environment on the stack (rule $[\widehat{\text{CEA}}]$). If q is a heap reference, we are calling a continuation that may have escaped.[1] 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 $\hat{c}$.

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

```
(call/cc (λ (c) (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

```
(λ1(f cc) (f (λ2(x k2) (cc x)) cc))
```

The CPS translation of its argument is

```
(λ3(c k) (c 42 (λ4(u) (somefun_CPS u k))))
```

The initial state $(\lambda_1,\ \{\lambda_3\},\ halt,\ \langle\rangle,\ \emptyset)$ is a $\widehat{UApply}$. (We abbreviate lambda

[1] We know that q was captured in a user closure, but we are not tracking whether the closure has been passed upward, so the continuation may still be on the stack. We pessimistically assume that it has escaped. More on that in section 5.4.

$$
\begin{array}{rcl}
\hat{\varsigma} \in \widehat{Eval} & = & Call \times Stack \times Heap \\
\hat{\varsigma} \in \widehat{UApply} & = & ULam \times \widehat{UProc} \times \widehat{CProc} \times Stack \times Heap \\
\hat{\varsigma} \in \widehat{CApply} & = & \widehat{CProc} \times \widehat{UProc} \times Stack \times Heap \\
\hat{d} \in \widehat{UProc} & = & Pow(ULam) \\
\hat{c} \in \widehat{CProc} & = & CLam + \{halt\} \\
fr, tf \in Frame & = & Var \rightharpoonup (\widehat{UProc} + \widehat{CProc}) \\
st \in Stack & = & Frame^* \\
h \in Heap & = & Var \rightharpoonup (\widehat{UProc} + Pow(\widehat{CProc} \times Stack))
\end{array}
$$

(a) Domains

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

$$
\begin{array}{ll}
[\widehat{\mathrm{UEA}}] & (\llbracket (f\ e\ q)^l \rrbracket, st, h) \rightsquigarrow (ulam, \hat{\mathcal{A}}_{\mathrm{u}}(e, l, st, h), \hat{c}, st', h) \\
& ulam \in \hat{\mathcal{A}}_{\mathrm{u}}(f, l, st, h) \\
& (\hat{c}, st') = \begin{cases} (q, st) & Lam_?(q) \\ (st(q), pop(st)) & Var_?(q) \end{cases}
\end{array}
$$

$$
\begin{array}{ll}
[\widehat{\mathrm{UAE}}] & (\llbracket (\lambda_l (u\ k)\ call) \rrbracket, \hat{d}, \hat{c}, st, h) \rightsquigarrow (call, st', h') \\
& st' = push([u \mapsto \hat{d},\ k \mapsto \hat{c}\,], st) \\
& h'(x) = \begin{cases} h(u) \cup \hat{d} & (x = u) \wedge H_?(u) \\ h(k) \cup \{(\hat{c}, st)\} & (x = k) \wedge H_?(k) \\ h(x) & o/w \end{cases}
\end{array}
$$

$$
\begin{array}{ll}
[\widehat{\mathrm{CEA}}] & (\llbracket (q\ e)^\gamma \rrbracket, st, h) \rightsquigarrow (\hat{c}, \hat{\mathcal{A}}_{\mathrm{u}}(e, \gamma, st, h), st', h) \\
& (\hat{c}, st') \in \begin{cases} \{(q, st)\} & Lam_?(q) \\ \{(st(q), pop(st))\} & S_?(\gamma, q) \\ h(q) & H_?(\gamma, q) \end{cases}
\end{array}
$$

$$
\begin{array}{ll}
[\widehat{\mathrm{CAE}}] & (\llbracket (\lambda_\gamma (u)\ call) \rrbracket, \hat{d}, st, h) \rightsquigarrow (call, st', h') \\
& st' = st[u \mapsto \hat{d}\,] \\
& h'(x) = \begin{cases} h(u) \cup \hat{d} & (x = u) \wedge H_?(u) \\ h(x) & o/w \end{cases}
\end{array}
$$

(b) Semantics

Figure 5.1: Abstract semantics of CFA2 with first-class control

expressions by labeled lambdas.) 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, producing a heap h with a single element $[\texttt{cc} \mapsto \{(halt, \langle\rangle)\}]$, and $\widehat{UEval}$ state

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

λ_2 is essentially a continuation reified as a user value. We tail call to λ_3, so we pop the stack, producing $\widehat{UApply}$ state

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

We next 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:

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

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

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

As `cc` is a heap reference, we ignore the current continuation and stack and restore $(halt, \langle\rangle)$ from the heap:

$$(halt,\ \{\texttt{42}\},\ \langle\rangle,\ h).$$

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

5.2.1 Correctness

To show that the abstract semantics for RCPS simulates the concrete semantics, we take the same steps as before.

The abstraction map $|\cdot|_{\text{ca}}$ is almost the same as the one from section 4.2.1. The only difference is when abstracting a variable environment ve to a heap h, because now we also keep the heap continuation variables. Each continuation variable k in ve is bound to a set of continuation-stack pairs in h. For each closure that can flow to k, we create a pair with the lambda of that closure and the corresponding stack.

$$|ve|_{\text{ca}} = \{\, (u, \bigcup_{(u,t)\in\text{dom}(ve)} |ve(u,t)|_{\text{ca}}) \;:\; (u \in UVar) \wedge H_?(u)\} \cup \\ \{\, (k, \bigcup_{(k,t)\in\text{dom}(ve)} makecs(ve(k,t), ve)) \;:\; (k \in CVar) \wedge H_?(k)\}$$

$$makecs(c, ve) \triangleq \begin{cases} (halt, \langle\rangle) & c = halt \\ (clam, toStack(LV(\mathcal{L}(clam)), \rho, ve)) & c = \langle clam, \rho\rangle \end{cases}$$

We also extend the ordering relation $\sqsubseteq$ to handle the continuation-stack pairs in the heap.

$$h_1(k) \sqsubseteq h_2(k) \quad \text{iff} \quad \forall\, p_1 \in h_1(k).\, \exists\, p_2 \in h_2(k).\, p_1 \sqsubseteq p_2$$

Note that $h_1(k)$ need not be a subset of $h_2(k)$; we only require that for each pair in $h_1(k)$ there is a more approximate pair in $h_2(k)$.

These are the only changes needed for $|\cdot|_{\text{ca}}$ and $\sqsubseteq$. The proof of the simulation theorem is included in appendix A.

5.3 Summarization for first-class control

Pushdown-reachability algorithms work on transition systems whose stack is unbounded, but the rest of the components are bounded. These algorithms get around the infinite state space by using summaries to weave calls and returns together. Due to escaping continuations, we also have to deal with infinitely many heaps. In this section, we show how to generalize summarization to do so.

Perhaps surprisingly, even though continuations can escape to the heap in the abstract semantics, we do not need continuations in the heaps of local states. Thus, we use the local semantics from section 4.3.2 unchanged. The abstraction function $|\cdot|_{\text{al}}$ drops the continuations from the abstract heaps.

$$|h|_{\text{al}} = h \upharpoonright UVar$$

Definition 10 gives a syntactic classification of states, which is used by the summarization algorithm. A $CEval$ state where the operator is a heap reference is different from an Exit-Ret because we make a non-local jump. We call such a state an **Exit-Esc** and treat it specially in the algorithm.

5.3.1 Workset algorithm

The algorithm is shown in fig. 5.2. We maintain two sets related to first-class continuations. If the continuation parameter of a user lambda is a heap variable, entries over that lambda are stored in $EntriesEsc$. $Escapes$ contains Exit-Esc states.

`Propagate` takes an extra argument which, if true, causes the propagated edge to be inserted in $Summary$ (lines 44 – 45). `Final` is the same as before so we omit it. `Update` is also unchanged, except the call to `Propagate`, which needs an extra argument (line 50).

Summaries for first-class continuations We handle escaping continuations using summaries. Consider the example from section 5.2. When control reaches `(cc x)`, we want to find which continuation flows to `cc`. We know that def_λ(`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 $def_\lambda(k)$.

Not all edges whose target is an Exit-Esc are summaries. The general form of a user lambda that binds a heap continuation variable is

`(`λ_1`(u k) (...(`λ_2`(u2 k2) (...(k e)...))...))`

where λ_1 contains a user lambda λ_2, which in turn contains a heap reference to `k` in operator position. To reach `(k e)`, the algorithm must go through an entry over λ_2. The path edge from that entry to the Exit-Esc state over `(k e)` should not be treated as a summary because we do not want the continuation that flows to `k2` to flow to `k`.

So, when the algorithm pulls an edge $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ from W where $\tilde{\varsigma}_2$ is an Exit-Esc, it must have a way to know whether to treat it as a summary or not. Let $(k\ e)^\gamma$ be the call site in $\tilde{\varsigma}_2$. The first solution that comes to mind is to consider $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ a summary iff $\tilde{\varsigma}_1$ is an entry over $def_\lambda(k)$. However, this will not work. Consider a tail call to a lambda whose continuation parameter is a heap variable.

```
(define f (λ1(u1 k1) ((λ2(u2 k2) (k1 u2)) u1 k1)))

((λ3(u3 k3) (f u3 k3)) 123 halt)
```

```
1   Summary, Callers, TCallers, EntriesEsc, Escapes, Final ← ∅
2   Seen, W ← {(Ĩ(pr), Ĩ(pr))}
3   while W ≠ ∅
4    remove (ς̃1, ς̃2) from W
5    switch ς̃2
6     case ς̃2 of Entry
7       for the ς̃3 in succ(ς̃2), Propagate(ς̃1,ς̃3,false)
8       ς̃2 of the form ([[(λl(u k) call)]], d̂, h)
9       if H?(k) then
10        insert ς̃2 in EntriesEsc
11        foreach ς̃3 in Escapes that calls k, Propagate(ς̃2,ς̃3,true)
12    case ς̃2 of CApply, Inner-CEval
13      for the ς̃3 in succ(ς̃2), Propagate(ς̃1,ς̃3,false)
14    case ς̃2 of Call
15      foreach ς̃3 in succ(ς̃2)
16        Propagate(ς̃3,ς̃3,false)
17        insert (ς̃1,ς̃2,ς̃3) in Callers
18        foreach (ς̃3,ς̃4) in Summary, Update(ς̃1,ς̃2,ς̃4)
19    case ς̃2 of Exit-Ret
20      if ς̃1 = Ĩ(pr) then Final(ς̃2)
21      else
22        insert (ς̃1,ς̃2) in Summary
23        foreach (ς̃3,ς̃4,ς̃1) in Callers, Update(ς̃3,ς̃4,ς̃2)
24        foreach (ς̃3,ς̃4,ς̃1) in TCallers, Propagate(ς̃3,ς̃2,false)
25    case ς̃2 of Exit-Esc
26      if (ς̃1,ς̃2) not in Summary then
27        insert ς̃2 in Escapes
28        ς̃2 of the form ([[(k e)^γ]], tf, h)
29        foreach ς̃3 in EntriesEsc over def_λ(k), Propagate(ς̃3,ς̃2,true)
30      else if ς̃1 = Ĩ(pr) then Final(ς̃2)
31      else
32        foreach (ς̃3,ς̃4,ς̃1) in Callers, Update(ς̃3,ς̃4,ς̃2)
33        foreach (ς̃3,ς̃4,ς̃1) in TCallers, Propagate(ς̃3,ς̃2,true)
34    case ς̃2 of Exit-TC
35      foreach ς̃3 in succ(ς̃2)
36        Propagate(ς̃3,ς̃3,false)
37        insert (ς̃1,ς̃2,ς̃3) in TCallers
38        S ← ∅
39        foreach (ς̃3,ς̃4) in Summary
40          insert (ς̃1,ς̃4) in S
41          Propagate(ς̃1,ς̃4,false)
42        Summary ← Summary ∪ S
43
44  Propagate(ς̃1, ς̃2, esc) ≜
45    if esc then insert (ς̃1,ς̃2) in Summary
46    if (ς̃1,ς̃2) not in Seen then insert (ς̃1,ς̃2) in Seen and W
47
48  Update(ς̃1, ς̃2, ς̃3) ≜
49    ...
50    Propagate(ς̃1,ς̃,false)
```

Figure 5.2: Workset algorithm

Here, we tail call λ_1 at call site `(f u3 k3)`. The algorithm will create a transitive summary from an entry over λ_3 to an Exit-Esc over `(k1 u2)` in order to make *halt* get called at `(k1 u2)`. We want this edge to be a summary even though its source is not an entry over $def_\lambda(\texttt{k1})$, which is λ_1.

We maintain an invariant for edges to Exit-Esc states: when the algorithm pulls such an edge out of W, the edge is treated as a summary only if it is in the $Summary$ set at that time.

Edge processing Each edge $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ is processed in one of six ways, depending on $\tilde{\varsigma}_2$. The treatment for returns, inner states (line 12), calls (line 14) and Exit-Ret states (line 19) is the same as before.

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).

5.3.2 Soundness

The algorithm terminates for the same reasons as the previous algorithm: there are finitely many edges and no edge is inserted in W twice.

For soundness, it suffices to show that for each reachable abstract state $\hat{\varsigma}$, the algorithm visits $|\hat{\varsigma}|_{\text{al}}$. However, first-class continuations create an intricate call/return structure, which complicates reasoning about soundness. When calls and returns nest properly, execution paths satisfy the *unique decomposition* property [59]: for each state $\hat{\varsigma}$ in the path, we can uniquely identify a state $\hat{\varsigma}'$ as the entry of the procedure that contains $\hat{\varsigma}$ (see def. 11, def. 12, lemma 21).

But 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

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

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.

Definition 16 (Corresponding Entries)**.**
For state $\hat{\varsigma}$ and path p, $CE_p(\hat{\varsigma})$ is the smallest set such that:

1. if $\hat{\varsigma}$ is an entry, $CE_p(\hat{\varsigma}) = \{\hat{\varsigma}\}$
2. if $p = p_1 \leadsto \hat{\varsigma}_1 \leadsto^+ \hat{\varsigma} \leadsto p_2$, $\hat{\varsigma}$ is an Exit-Esc over $(k\ e)^\gamma$, $\hat{\varsigma}_1$ is an entry over $def_\lambda(k)$, then $\hat{\varsigma}_1 \in CE_p(\hat{\varsigma})$.
3. if $p = p_1 \leadsto \hat{\varsigma}_1 \leadsto \hat{\varsigma} \leadsto p_2$, $\hat{\varsigma}$ is neither an entry nor an Exit-Esc, $\hat{\varsigma}_1$ is neither an Exit-Ret nor an Exit-Esc, then $CE_p(\hat{\varsigma}) = CE_p(\hat{\varsigma}_1)$.

4. if $p = p_1 \rightsquigarrow \hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^* \hat{\varsigma}_3 \rightsquigarrow^+ \hat{\varsigma}_4 \rightsquigarrow \hat{\varsigma} \rightsquigarrow p_2$, $\hat{\varsigma}$ 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 $CE_p(\hat{\varsigma}_1) \subseteq CE_p(\hat{\varsigma})$.
5. if $p = p_1 \rightsquigarrow \hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^+ \hat{\varsigma}_3 \rightsquigarrow \hat{\varsigma} \rightsquigarrow p_2$, $\hat{\varsigma}$ 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 $CE_p(\hat{\varsigma}_1) \subseteq CE_p(\hat{\varsigma})$.

Items 1, 3 and 5 correspond to the cases of def. 11. However, one might have expected that in item 5 we would have $CE_p(\hat{\varsigma}_1) = CE_p(\hat{\varsigma})$ instead of $CE_p(\hat{\varsigma}_1) \subseteq CE_p(\hat{\varsigma})$. But because of escaping continuations, $CE_p^*(\hat{\varsigma}_3)$ can contain multiple entries whose predecessor is a call, so we should allow all these flows.

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 $iu_\lambda([\![(k\ e)^\gamma]\!]) \neq def_\lambda(k)$. (Remember also that the workset algorithm does not consider the edge $(|\hat{\varsigma}'|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ to be a summary.) For all other states, $CE_p(\hat{\varsigma})$ is the set of entries of procedures that contain $\hat{\varsigma}$.

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 17. For state $\hat{\varsigma}$ and path p, $CE_p^*(\hat{\varsigma})$ is the smallest set such that:

- $CE_p(\hat{\varsigma}) \subseteq CE_p^*(\hat{\varsigma})$
- if $p = p_1 \rightsquigarrow \hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^* \hat{\varsigma} \rightsquigarrow p_2$, $\hat{\varsigma}_2 \in CE_p(\hat{\varsigma})$, $\hat{\varsigma}_1$ is a Tail Call, then $CE_p^*(\hat{\varsigma}_1) \subseteq CE_p^*(\hat{\varsigma})$.

Theorem 18 (Soundness)**.** Let $p = \hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}$. Then, after summarization:

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

The proof of the soundness theorem can be found in appendix A.

5.3.3 Incompleteness

CFA2 without first-class control is complete, so there is no loss in precision when going from abstract to local states. The algorithm of fig. 5.2 is not complete; it may compute flows that never happen in the abstract semantics. Consider the code:

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

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

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

In this program, `esc` is the CPS translation of `call/cc`. The two user functions λ_1 and λ_2 expect a continuation reified as a user value 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 finds that $\{$`"foo","bar"`$\}$ flows to `b`. At the second call to `esc`, it spuriously connects the entry to the Exit-Esc state over `(cc x)` at line 11 (fig. 5.2).

In previous chapters, we saw that kCFA approximates a program as a finite-state automaton and CFA2 for CPS\1 approximates a program as a PDS. What kind of transition system does the abstract semantics for RCPS describe? We cannot translate the semantics to a PDS in the same way as before, because heaps were part of the control states and now we have infinitely many heaps.

It seems likely that the abstract semantics corresponds to a machine M strictly more expressive than PDSs. Fortunately, not knowing what M is did not stop us from creating a computable overapproximation of it, which is a testament to the generality of abstract interpretation and operational semantics.

5.4 Variants for downward continuations

A user lambda that binds a heap continuation variable has the form

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

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

can avoid merging.

One way to analyze exceptions precisely is by uniformly passing two continuations to each user function, the current continuation and an exception handler [4]. Consider a lambda $(\lambda$ `(u k1 k2)` $(\ldots($`k2 e`$)^{\gamma}\ \ldots))$ where $S_?(\gamma,$ `k2`$)$ holds. Every Exit-Ret over $($`k2 e`$)^{\gamma}$ is an exception throw. The handler continuation lives somewhere on the stack. To find it, we propagate transitive summaries for calls, as 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 $($`k2 e`$)^{\gamma}$, it searches in $\mathit{Callers}$ 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, so as not to 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 section. During flow analysis, we track if any closure over λ_2 escapes upward. We do this 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 $\tilde{\varsigma}_2$, then λ_2 is passed upward and we look up `k` at `(k e)` in the heap. 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 `(k e)`, 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`.

CHAPTER 6

Pushdown flow analysis using big-step semantics

This chapter presents another approach to pushdown higher-order flow analysis. We write an abstract interpreter that resembles a traditional interpreter: it is a collection of mutually recursive functions, using big-step semantics [23]. We call this analysis Big CFA2.

Big CFA2 is broadly applicable, but less so than CFA2; it applies to typed and untyped languages, with first-class functions, side effects and exceptions, but does not handle first-class control. Big CFA2 is designed for scalability. It minimizes caching and comparing of abstract states, so it is fast and uses little memory. Also, it finds return points in constant time, without maintaining a $Callers$ set.

Section 6.1 provides motivation for Big CFA2. Section 6.2 describes the core algorithm. Sections 6.3, 6.4 and 6.5 describe three extensions to the core algorithm. In section 6.3, we show how to analyze exceptions precisely by incorporating them in summaries. Section 6.4 discusses mutable state. In section 6.5, we show how to use exceptions in the implementation language to avoid running out of stack. These extensions are orthogonal and can be implemented on top of each other. For simplicity, we present them as independent additions to the algorithm of section 6.2. We believe that Big CFA2 is sound, but have not formally proved soundness. Instead, we argue the correctness of the algorithms informally.

```
1  W ← {init}
2  Seen ← {init}
3  while W ≠ ∅
4    remove elm from W
5    for every successor elm2 of elm
6      if (elm2 ∉ Seen) then
7        insert elm2 in Seen and W
```

Figure 6.1: General structure of an iterative flow analysis

6.1 Iterative flow analyses

Flow analyses based on abstract interpretation are usually implemented *iteratively* using a workset. Fig. 6.1 shows the high-level structure of an iterative analysis. The algorithms for kCFA and CFA2 have this form (see fig. 3.3, 4.8, 5.2). (The workset elements in kCFA are states, whereas in CFA2 they are pairs of states.) As we mentioned in sec. 2.2, an iterative analysis requires that all subexpressions of the program be named. During the analysis, every subexpression causes the creation of one or more abstract states.

We claim that this fine granularity creates extra work for the analysis. Suppose that we want to analyze a call to a function f with argument a in environment ρ. For every subexpression in the body of f, an iterative analysis searches through `Seen` and compares abstract states for equality—both expensive operations (line 6).

We make an optimistic assumption: if we have not seen this call to f before, most states in the body of f will be new. Instead of doing the expensive search and comparison at every step, we can analyze the whole body blindly, without caching intermediate results.

6.2 Big CFA2

6.2.1 Syntax and preliminary definitions

Fig. 6.2 shows the target language for Big CFA2. It is an untyped λ-calculus with numbers and a few operations on numbers. We choose an extended λ-calculus because the big-step structure of the analysis stands out more if the target language has more expressions than just lambdas and calls.

$$
\begin{array}{lcl}
e & ::= & x \mid n \mid \texttt{(}\lambda\texttt{(}x\texttt{)}\ e\texttt{)} \mid \texttt{(}e_1\ e_2\texttt{)} \mid \texttt{(ifz}\ e_1\ e_2\ e_3\texttt{)} \mid \texttt{(}e_1\ op\ e_2\texttt{)} \\
op & ::= & \texttt{+} \mid \texttt{-} \mid \ldots
\end{array}
$$

Figure 6.2: Syntax

$$
\begin{array}{rcl}
Val & = & Pow(\{\mathbb{R}\} \cup Lam) \\
Heap & = & Var \rightharpoonup Val \\
Frame & = & Var \rightharpoonup Val \\
Summary & = & Lab \times Time \times Val \rightharpoonup Val
\end{array}
$$

Figure 6.3: Abstract domains

The values in the language are numbers and closures. The call-by-value concrete semantics for numbers, variable lookup, lambdas and calls is standard. For an `(ifz` e_1 e_2 e_3`)` expression, we first evaluate e_1 to a value v. If v is 0, we evaluate e_2. If it is a non-zero number, we evaluate e_3. It is an error if v is not a number. The expression `(`e_1 op e_2`)` expects two numeric operands and gives back a number; op can be any of the usual arithmetic operators.

Fig. 6.3 shows the abstract domains. Lam is the finite set of lambda expressions in the program. Numbers abstract to a single abstract number $\mathbb{R}$. An abstract value is a subset of $\{\mathbb{R}\} \cup Lam$. For scalability, we use Shivers's timestamp technique (see sec. 4.6.1). A summary is a pair $((l, t, v), v')$ of a function entry and an abstract value, meaning that if the function labeled l is called with argument v at time t, it returns an abstract value v'.

6.2.2 Abstract interpreter

The core algorithm for Big CFA2 appears in fig. 6.4. We assume that all variables in the analyzed program have unique names and every variable reference is labeled as a stack or a heap reference. Initially, the global variable `timestamp` is 0 and `heap` and `Summary` are empty.

`EvalExp` takes an expression e and a frame `fr` and returns an abstract value, which is the result of evaluating e in the environment `fr` (and, of course, the global environment `heap`). We start the analysis by calling `EvalExp` with a closed expression to be analyzed and an empty frame.

`EvalExp` branches depending on the kind of the input expression e. If e is a number, we return the singleton set $\{\mathbb{R}\}$ (line 9). If e is a lambda, we

return the singleton set containing that lambda (line 11).

If e is a variable, we look it up in `fr` if it is a stack reference and in `heap` otherwise (lines 12 – 15). For our small language with single-argument functions, each frame contains only one binding. In practical languages, functions are multi-arity and there are local variable declarations, so frames contain many bindings. However, an important difference from CFA2 is that we no longer use frames for the results of intermediate computations, we consume these results immediately.

If e is an arithmetic operation, we evaluate the operands and return $\{\mathbb{R}\}$ (lines 16 – 19). For convenience, we ignore runtime errors like adding a number to a function. We discuss exceptions in sec. 6.3.

For `ifz` expressions, we first evaluate the test e_1 (line 21). Since all numbers abstract to a single abstract number, we cannot know if e_1 evaluates to 0, so we analyze both branches and join the results (line 22).

For a call $(e_1\ e_2)$, we first evaluate the operator and the operand. We call every lambda that can flow to e_1 with `av2` and return the join of the results (lines 23 – 30).

The function `ApplyFun` is used to analyze a call to a function $(\lambda_l(x)\ e)$ with abstract value `av` as the argument. `ApplyFun` returns an abstract value, which is the result of the call. We borrow the `var` notation from JavaScript to declare local variables (line 33). In line 34, we check if we have seen a call to λ_l with the same argument and the same heap. If we find a summary, we return immediately (line 36). Otherwise, we must analyze the call.

If the formal parameter x is a heap variable, we must update its value in the heap (line 38). If `av` contains things that have not yet flowed to x, we join it with the heap value of x. Since the heap changes, we increase the timestamp (lines 2 – 4).

We record the timestamp at the beginning of the call because it may increase during the call (line 39). Then, we create a frame that binds x to `av` and evaluate the body of λ_l (line 40). When the call finishes, we record a summary and return the result (lines 41, 42).

This algorithm exhibits several desirable properties. First, it minimizes caching-and-comparing of abstract states. It does not require labeling every subexpression in the program; it analyzes the direct-style AST as is, and it uses the results of subexpressions immediately. Also, when it enters a function for which there is no summary, it assumes that the abstract states

```
1   UpdateHeap(x, av) ≜
2     if (av ⋢ heap[x]) then
3       timestamp++
4       heap[x] ← heap[x] ⊔ av
5
6   EvalExp(e, fr) ≜
7     switch e
8       case n
9         return {ℝ}
10      case ⟦(λ(x) e1)⟧
11        return {⟦(λ(x) e1)⟧}
12      case x^l
13        if S?(l, x) then
14          return fr[x]
15        return heap[x]
16      case ⟦(e1 op e2)⟧
17        EvalExp(e1, fr)
18        EvalExp(e2, fr)
19        return {ℝ}
20      case ⟦(ifz e1 e2 e3)⟧
21        EvalExp(e1, fr)
22        return EvalExp(e2, fr) ⊔ EvalExp(e3, fr)
23      case ⟦(e1 e2)⟧
24        let av1 ← EvalExp(e1, fr)
25            av2 ← EvalExp(e2, fr)
26            res ← ∅
27        in
28          foreach ⟦(λ(x) e)⟧ in av1
29            res ← res ⊔ ApplyFun(⟦(λ(x) e)⟧, av2)
30          return res
31
32  ApplyFun(⟦(λl(x) e)⟧, av) ≜
33    var tStart, retv, summ
34    summ ← Summary[l, timestamp, av]
35    if (summ ≠ undefined) then
36      return summ
37    if H?(x) then
38      UpdateHeap(x, av)
39    tStart ← timestamp
40    retv ← EvalExp(e, [x ↦ av])
41    Summary[l, tStart, av] ← retv
42    return retv
```

Figure 6.4: The core algorithm

in the body are likely to be new. Therefore, unlike iterative analyses, it does not search the visited set and compare states for equality at every step (*cf.* fig. 6.1, line 6).

Second, Big CFA2 finds return points in constant time, without maintaining a $Callers$ set. The call structure of the algorithm follows the call structure of the analyzed program, so the result of a call can simply be passed to the context (line 30). Intuitively, the algorithm uses the runtime stack of the implementation language as a data structure to replace $Callers$. The iterative CFA2 has to search $Callers$ and compare abstract states for equality (*cf.* fig. 4.8, line 18).

Last, we avoid "tail duplication" (sec. 4.6.1) by design, because we join after a control-flow split, in lines 22 and 29.

6.2.3 Analysis of recursive programs

The algorithm of fig. 6.4 does not terminate when the analyzed program is recursive. When a function entry eventually leads to itself, the second time we see it we do not have a summary for it yet, so we keep going.

Fig. 6.5 shows how to modify `ApplyFun` to solve this problem. During the analysis, we maintain a map called $Pending$ to keep track of entries for function calls that have not returned.

$$pen \in Pending = Lab \to Pow(Time \times Val)$$

For every function λ_l, if (t, v) is in $pen(l)$ then there is a call to λ_l that we have not finished analyzing; the argument passed to λ_l is v and the timestamp at the time of the call is t. Intuitively, we can think of $Pending$ as a set of entries.

We maintain the following invariant: for every `ApplyFun` frame in the runtime stack, there is an element in $Pending$. The size of $Pending$ is bounded. Therefore, the size of the runtime stack is also bounded.

The local variable `retv` holds the result of a fixpoint computation. We initialize it to $\emptyset$, which is the bottom abstract value (line 3). We turn lines 39 – 42 of fig. 6.4 to a fixpoint computation (lines 9 – 23). If a pending entry is equal to the current entry, we have recurred. We create a summary that maps the current entry to $\emptyset$ and return $\emptyset$ (lines 11 – 13). The idea is that a non-terminating recursion returns nothing, *i.e.*, bottom. For a recursion to eventually terminate with a result, there must be a base case somewhere.

```
1   ApplyFun([[(λ_l(x) e)]], av) ≜
2     var tStart, retv, summ
3     retv ← ∅
4     summ ← Summary[l, timestamp, av]
5     if (summ ≠ undefined) then
6       return summ
7     if H_?(x) then
8       UpdateHeap(x, av)
9     while true
10      tStart ← timestamp
11      if ((tStart, av) ∈ Pending[l]) then
12        Summary[l, tStart, av] ← ∅
13        return ∅
14      insert (tStart, av) in Pending[l]
15      retv ← retv ⊔ EvalExp(e, [x ↦ av])
16      remove (tStart, av) from Pending[l]
17      summ ← Summary[l, tStart, av]
18      if (summ = undefined) then
19        Summary[l, tStart, av] ← retv
20        return retv
21      if (summ = retv) then
22        return retv
23      Summary[l, tStart, av] ← retv
```

Figure 6.5: Fixpoint computation for recursive functions

If the current entry is fresh, we put it in `Pending`. Then, we analyze the body and update `retv`. After the call, we remove the entry from `Pending` (lines 14 – 16). There are three cases. If the call was not recursive, we do not have a summary for it, so we create one and return (lines 18 – 20). If it was recursive and the return value of the stored summary is equal to `retv` (line 21), then we have reached a fixpoint, so we stop. If the call was recursive and `retv` is not equal to `summ`, then we update the summary and loop (line 23). Note that, during the `while` loop, we maintain the invariant that `summ` $\sqsubseteq$ `retv` (lines 12, 19, 23).

The operations on `Pending` can be implemented efficiently. Remember that every entry in `Pending` corresponds to an `ApplyFun` frame. The frames of `ApplyFun` on the stack have the form

(l_1, t_1, v_1)	(l_2, t_2, v_2)	...	(l_n, t_n, v_n)

where each (l_i, t_i, v_i) is a call to λ_{l_i} with argument v_i and the value of `tStart` is t_i. (The analyzed program may be recursive, so the l_is need not be distinct.) Because the timestamp can only increase, we get $t_1 \leqslant t_2 \leqslant \ldots \leqslant t_n$.

For a function λ_l, the elements of `Pending[`l`]` are inserted and removed in a FIFO manner (lines 14, 16). Therefore, we can implement `Pending[`l`]` with a stack s. This way, insertion and removal take constant time. Moreover, if s contains k pairs,

(t_1, v_1)	(t_2, v_2)	...	(t_k, v_k)

we know that $t_1 \leqslant t_2 \leqslant \ldots \leqslant t_k$. So, at line 11, we can start searching at the top of s. If we do not find a matching pair, we stop when the timestamp becomes smaller.

6.2.4 Discarding deprecated summaries to save memory

The size of `Summary` can grow quickly because new summaries are constantly added as the heap grows and the timestamp increases. In this section, we show how to keep the memory usage for `Summary` low.

Consider a function `f`, which is part of some larger JavaScript program.

```
1  function f(x) {
2    ...
3    g(e2);
4    ...
5  }
```

Assume we are analyzing this program and we find a call to `f`. We call `ApplyFun` to analyze the call. The timestamp has some value t. While analyzing the body of `f` or the call to `g` (lines 2, 3) it is likely that the heap will be modified, so the timestamp will increase. After we return from `g`, we analyze the rest of `f` and create a summary. *This summary is unnecessary; it will never be used in the rest of the analysis.* The summary contains the older timestamp t. But when `ApplyFun` searches for a summary, it uses the *current* value of the timestamp (fig. 6.5, line 4).

Each timestamp increase deprecates all summaries with a smaller timestamp. We implement `Summary` as a hash table with function labels as keys. Thus, the bucket for a label l contains all summaries for λ_l. We maintain the following invariant: all summaries in a bucket have the same timestamp. When we insert a summary $((l, t, v), v')$, if the other summaries for λ_l have a smaller timestamp, we discard them.

```
1    ApplyFun([[(λl(x) stm)]], av) ≜
2      ...
3      fr ← [x ↦ av]
4      W ← {stm}
5      while W ≠ ∅
6        remove a statement s from W
7        if (s is [[return e;]]) then
8          Call EvalExp(e, fr) and join the result with retv
9        else
10         Call EvalStm(s, fr) and add the successors of s to W
11     ...
```

Figure 6.6: Pseudocode for the analysis of statements

6.2.5 Analysis of statements

Most mainstream higher-order languages are not expression-based like Lisp, Scheme and ML. They also have statements and use C-style syntax. Here, we sketch briefly how to extend Big CFA2 for languages with statements.

In languages with C-style syntax, a statement cannot be a subterm of an expression, but an expression can be a subterm of a statement. Therefore, we can analyze expressions recursively and statements iteratively.

Before flow analysis, we process the AST and connect each statement to its successors to form a control-flow graph. Branch statements such as `if` and `switch` have more than one successor.

The body of a function $(\lambda_l(x)\ stm)$ is now a block statement stm. In `ApplyFun`, we use a workset for statements. We turn line 15 in fig. 6.5 to an iteration that uses the workset. Fig. 6.6 shows the pseudocode. The function `EvalStm` analyzes a statement and returns its successors.

Even though we are using a workset for statements, we are not caching analysis results for each statement. A statement contributes to the analysis by influencing the return value `retv` and by causing changes in `heap`.

6.3 Exceptions

Exceptions create a different control flow from normal function call and return. An exception may propagate down the stack, causing several frames to be popped until a handler is found. Also, a function may throw an exception if called in one calling context and return normally in another. In this section, we show how pushdown analyses can deal with exceptions with the

```
1   EvalExp(e, fr) ≜
2     switch e
3       case ⟦(try e1 catch e2)⟧
4         let (av1, exn1) ← EvalExp(e1, fr)
5         in
6           if exn1 then
7             let (av2, exn2) ← EvalExp(e2, fr)
8             in
9               return (av1 ⊔ av2, exn2)
10          return (av1, false)
11      case ⟦(ifz e1 e2 e3)⟧
12        let (av1, exn1) ← EvalExp(e1, fr)
13        in
14          if (lam ∈ av1) then
15            exn1 ← true
16          if (ℝ ∉ av1) then
17            return (∅, exn1)
18          let (av2, exn2) ← EvalExp(e2, fr)
19              (av3, exn3) ← EvalExp(e3, fr)
20          in
21            return (av2 ⊔ av3, exn1 or exn2 or exn3)
```

Figure 6.7: Changes to EvalExp for exceptions

same precision as normal return values; an exception is only propagated to the correct calling context. Our solution is not specific to Big CFA2, it also applies to CFA2 and any other summary-based analysis.

Our small dynamic language throws an exception if there is a type mismatch at runtime, like Scheme. For example, adding a number to a function is an error; when the test expression of `ifz` evaluates to a function, it is an error; applying a number is an error. These are the only errors that we will deal with.

We do not add an explicit `(throw` e`)` expression. Also, we use a single error value instead of multiple kinds of errors. These two features are useful in a real language, but not fundamental for an analysis. Our minimal exception system is expressive enough to demonstrate all important issues of exception handling.

We add an expression `(try` e_1 `catch` e_2`)` with the following concrete semantics: if e_1 evaluates to a value, it is the result of the expression; if e_1 throws the sole error value, the evaluation of e_2 determines the result of the expression.

EvalExp now returns a pair `(av, exn)` where `av` is the abstract value of e and `exn` is a boolean. If `exn` is true, e *may* throw an exception. Fig. 6.7

```
1   ApplyFun([[(λ_l(x) e)]], av) ≜
2     var tStart, retv, exn, retv2, exn2, summ
3     retv ← ∅
4     exn ← false
5     summ ← Summary[l, timestamp, av]
6     if (summ ≠ undefined) then
7       return summ
8     if H?(x) then
9       UpdateHeap(x, av)
10    while true
11      tStart ← timestamp
12      if ((tStart, av) ∈ Pending[l]) then
13        Summary[l, tStart, av] ← (∅, false)
14        return (∅, false)
15      insert (tStart, av) in Pending[l]
16      (retv2, exn2) ← EvalExp(e, [x ↦ av])
17      retv ← retv ⊔ retv2
18      exn ← exn or exn2
19      remove (tStart, av) from Pending[l]
20      summ ← Summary[l, tStart, av]
21      if (summ = undefined) then
22        Summary[l, tStart, av] ← (retv, exn)
23        return (retv, exn)
24      if (summ = (retv, exn)) then
25        return summ
26      Summary[l, tStart, av] ← (retv, exn)
```

Figure 6.8: Changes to `ApplyFun` for exceptions

shows how to analyze `try/catch` and how to change `ifz` for exceptions. The changes for the other expressions are similar.

When e is `(try` e_1 `catch` e_2`)`, we first evaluate e_1 (line 4). (We use pattern-matching to access the components of the pair.) If e_1 does not throw, e evaluates to `av1` (line 10). If `exn1` is true, e_1 may throw, so we evaluate e_2 (line 7). The value of e is `av1` $\sqcup$ `av2` and it throws iff e_2 throws (line 9).

When e is `(ifz` e_1 e_2 e_3`)`, we evaluate the test (line 12). If there is a lambda in `av1`, e_1 can throw, so we set `exn1` to `true` (line 15). Moreover, if $\mathbb{R}$ is not in `av1`, control cannot reach the branches of `ifz`, so we return (line 17). Otherwise, we evaluate both branches. If at least one of e_1, e_2, e_3 can throw, then e can throw (line 21).

If a function throws an exception, we can record that in a summary. *Summary* now maps entries to pairs of return values and booleans.

$$Summary = Lab \times Time \times Val \rightharpoonup Val \times Boolean$$

Fig. 6.8 shows the new `ApplyFun`. The changes are simple. `ApplyFun`

returns a pair of an abstract value and a boolean. We use the local variable `exn` to record if the function can throw an exception. The bottom summary for recursion is now `(∅, false)` (line 14). To decide if we have reached the fixpoint, we look at both the return value and the exception (line 24).

6.4 Mutation

In a language with mutable variables, stack and heap references interact. A change in the value of a heap reference may cause a change in the value of a stack reference and vice versa. Consider the following JavaScript snippet.

```
function f1(x1) {
  (function g1() { x1 = "foo"; })();
  return x1;
}

f1(5);

function f2(x2) {
  x2 = "foo";
  return (function g2() { return x2; })();
}

f2(44);
```

In `f1`, we assign to the heap reference of `x1` and the `return` uses a stack reference. A sound analysis should find that `x1` can be bound to a string in the frame. Similarly, in `f2`, a sound analysis should find that a string can flow to the heap reference of `x2`.

Suppose we add an expression $(x := e)$ in our language, with the following semantics: the result of evaluating the rvalue e is assigned to x and is also the result of the expression. How would we change Big CFA2? One solution is to change how we classify variable references: if a reference of a heap variable x is assigned, we classify *all* x references as heap references. This is not ideal because it sacrifices precision for all x references to ensure soundness.

In this section, we describe a solution that does not change the classification of references. We use timestamps in the frames and the heap to know when bindings are created or modified.

```
1   UpdateHeap(x, av) ≜
2     let (hav, hts) ← heap[x]
3     in
4       if (av ⋢ hav) then
5         timestamp++
6         heap[x] ← (hav ⊔ av, timestamp)
7
8   EvalExp(e, fr) ≜
9     switch e
10      case ⟦(x^l := e1)⟧
11        let (av, fr2) ← EvalExp(e1, fr)
12        in
13          if H?(l, x) then
14            UpdateHeap(x, av)
15            return (av, fr2)
16          if H?(x) then
17            UpdateHeap(x, av)
18          return (av, fr2[x ↦ (av, timestamp)])
19      case x^l
20        if S?(x) then
21          return ((car fr[x]), fr)
22        if H?(l, x) then
23          return ((car heap[x]), fr)
24        let (frav, frts) ← frame[x]
25            (hav, hts) ← heap[x]
26        in
27          if (frts < hts) then
28            return (hav, fr[x ↦ (hav, timestamp)])
29          return (frav, fr)
30      case (ifz e1 e2 e3)
31        let (av1, fr1) ← EvalExp(e1, fr)
32            (av2, fr2) ← EvalExp(e2, fr1)
33            (av3, fr3) ← EvalExp(e3, fr1)
34        in
35          return (av2 ⊔ av3, fr2 ⊔ fr3)
36
37  ApplyFun(⟦(λl(x) e)⟧, av) ≜
38    ...
39    retv ← retv ⊔ EvalExp(e, [x ↦ (av, tStart)])
40    ...
```

Figure 6.9: Changes to the code for mutation

$$Heap = Var \rightharpoonup (Val \times Time)$$
$$Frame = Var \rightharpoonup (Val \times Time)$$

Fig. 6.9 shows the new code. Evaluating an expression may change a binding in the frame, so `EvalExp` returns a pair of an abstract value and a new frame.

When e is `(`x^l `:=` e_1`)`, we evaluate the rvalue e_1 (line 11). If x^l is a heap reference, we must update the heap (line 14). If `UpdateHeap` changes the heap value of x, it marks the new binding with the current timestamp (line 6). In lines 16 – 18, x^l is a stack reference. The assignment will change the binding of x in the frame. Thus, the frame in the result indicates that x is bound to `av` and the binding happened at time `timestamp` (line 18). Moreover, if x is a heap variable, we call `UpdateHeap` because the assignment may impact the heap references of x (line 17). Generally, for any heap variable x, we maintain the invariant that the heap binding of x is more general than any of x's stack bindings.

When e is a variable reference x^l and x is a stack variable, we look it up in `fr` (line 21). In lines 22 – 29, x is a heap variable. If x^l is a heap reference, we look it up in the heap (line 23). The case when x^l is a stack reference of a heap variable is more involved. First, we look up x in both the frame and the heap (lines 24, 25). If the heap binding is fresher than the frame binding (line 27), x changed in the heap after we last bound it in the frame, so to be sound we update its frame binding with the heap value `hav` (line 28). Otherwise, we return the frame value (line 29).

Mutation requires that we make simple changes to the treatment of the other expressions. We only show the changes to `ifz`, the other cases are similar. We first evaluate the test e_1 (line 31). We then evaluate both branches in the new environment `fr1`. The branches can have different side effects, so we join `fr2` and `fr3` in the result (line 35).

As an aside, note that when an expression changes the frame, we create a new frame instead of mutating `fr`. Thus, expressions in the same function can be evaluated in different environments, which makes Big CFA2 flow sensitive. This choice is not fundamental for the analysis; one could modify `EvalExp` to get flow insensitivity.

`ApplyFun` stays practically unchanged. We only modify line 15 because frame bindings now contain timestamps (line 39).

6.5 Managing the stack size

Big CFA2 is deeply recursive. If it is implemented in a mainstream language with a fixed-size stack, it will run out of stack even for mid-sized programs. In this section, we show how to keep stack usage to a minimum.

We have mentioned that every live activation of `ApplyFun` corresponds to a function entry

(l_1, t_1, v_1)	(l_2, t_2, v_2)	...	(l_n, t_n, v_n)

and $t_1 \leqslant t_2 \leqslant \ldots \leqslant t_n$. If the current timestamp is t_n, every pending entry whose timestamp t_i is less than t_n is deprecated; it will lead to the creation of a summary that will not be used (sec. 6.2.4).

If the implementation language provides exceptions, *we can throw an exception to clear unneeded pending calls to* `ApplyFun` *off the stack*. Even though we cannot inspect the stack directly to find these calls, we can do it indirectly by looking in `Pending`. Suppose the entry in the most recent `ApplyFun` activation is (l, t, v). We list three ways to do stack cleaning, from least to most aggressive.

1. If there is a pending entry (l, t', v) where $t' < t$, clear the stack up to that entry. With this strategy, we only throw an exception when there is a pending call to the same function with the same argument.
2. If there is a pending entry (l, t', v') where $t' < t$, clear the stack up to that entry. With this strategy, we throw every time there is a pending call to the same function, regardless of the argument.
3. If there is a pending entry (l', t', v') where $t' < t$, clear the stack up to that entry. This strategy never allows a deprecated entry on the stack.

We implemented all three and settled for the second one. Strategy 3 is the slowest because by clearing the stack more often it causes more re-analysis. Strategies 1 and 2 perform about the same, but strategy 2 keeps the stack smaller. In the rest of this section, we show how to implement strategy 2.

`EvalExp` does not change. Fig. 6.10 shows the new `ApplyFun`. We wrap the body of the `while` loop in a `try` block (lines 11 – 27). Lines 12 – 14 are new: before we analyze the body of λ_l, we retrieve the set of pending entries for λ_l (line 12). If there is an entry whose timestamp is smaller than `tStart` (line 13), we want to clear the stack.

```
1  ApplyFun([[(λ_l(x) e)]], av) ≜
2    var tStart, retv, summ, pen
3    retv ← ∅
4    summ ← Summary[l, timestamp, av]
5    if (summ ≠ undefined) then
6      return summ
7    if H?(x) then
8      UpdateHeap(x, av)
9    while true
10     try {
11       tStart ← timestamp
12       pen ← Pending[l]
13       if (pen contains (ts,av2) such that ts < tStart) then
14         throw {label: l, numframes: |pen|, justthrown: true}
15       if ((tStart, av) ∈ pen) then
16         Summary[l, tStart, av] ← ∅
17         return ∅
18       insert (tStart, av) in Pending[l]
19       retv ← retv ⊔ EvalExp(e, [x ↦ av])
20       remove (tStart, av) from Pending[l]
21       summ ← Summary[l, tStart, av]
22       if (summ = undefined) then
23         Summary[l, tStart, av] ← retv
24         return retv
25       if (summ = retv) then
26         return retv
27       Summary[l, tStart, av] ← retv
28     }
29     catch (exn) {
30       if (exn.justthrown) then
31         exn.justthrown ← false
32         throw exn
33       remove (tStart, av) from Pending[l]
34       if (exn.label ≠ l) then
35         throw exn
36       if (exn.numframes ≠ 1) then
37         exn.numframes--
38         throw exn
39     }
```

Figure 6.10: Using exceptions to keep the stack small

At this point, there are |pen| frames of ApplyFun in the stack for λ_l besides the current one. There is no entry in Pending for the current frame because insertion happens at line 18. We know |pen| $\geqslant 1$ because the test at line 13 was true. The deepest of these frames is for an entry of the form (l, t, v) where $t <$ tStart. We want to throw to that frame and resume the analysis there. The thrown value is a record that contains the label of the function that caused the throw, the number of pending entries for that function and a boolean flag justthrown set to true (line 14).

The catch block is responsible for stopping the propagation of the exception at the right frame (lines 30 – 38). The variable exn is local to the catch body and gets bound to the thrown value.

If exn.justthrown is true, the exception was just thrown from the current activation of ApplyFun. Thus, we set the flag to false and propagate the exception down the stack (lines 30 – 32).

If exn.justthrown is false, the exception was thrown from another frame and propagated here. Lines 34 – 38 decide if we need to rethrow or if we resume the analysis here. In both cases, we remove the entry for the current ApplyFun activation from Pending (line 33) to maintain the invariant (see sec. 6.2.3).

The function that caused the throw is labeled exn.label. If the current frame is for another function, *i.e.*, exn.label $\neq l$, we rethrow (line 35). If this frame is for the right function, but it is not the deepest one, we decrease the count for pending frames and rethrow (lines 36 – 38). If exn.numframes is 1, we must resume the analysis here. Control leaves the catch block and goes back to the start of the while loop. By clearing the stack, Pending[l] is now $\emptyset$ so control will reach line 19 and continue the analysis.

CHAPTER 7

Building a static analysis for JavaScript

In this chapter, we report on the implementation of DoctorJS, a static-analysis tool for JavaScript based on Big CFA2.[1] Our primary purpose is to highlight the trade-offs involved when building a static analysis for full JavaScript. For some language constructs, we present several possible abstractions, each with its own pros and cons.

JavaScript has only one kind of composite data: the object. Functions, arrays and regular expressions are all objects. The language compensates for this lack of variety by making objects extremely flexible. Objects are maps from property names to values. The number of properties in an object is not fixed: properties can be added to and deleted from objects at runtime. The variable object size combined with inheritance make static analysis difficult, because adding (*resp.* removing) properties can shadow (*resp.* expose) inherited properties.

There are three basic requirements of any static analysis: speed, precision and soundness. We do not know how to satisfy all three for JavaScript. DoctorJS leans toward speed and precision and allows false negatives in some cases.

Section 7.1 gives some necessary background on the JavaScript object model. In section 7.2, we describe how DoctorJS handles various JavaScript constructs. We mainly discuss the difficulties related to the analysis of objects. We omit the discussion of some interesting but less prominent features, such as `eval`, `with`, the `arguments` array and property attributes (enumerable, writable, configurable).

[1] Big CFA2 did not predate DoctorJS. DoctorJS was the result of the author's effort to apply the ideas in CFA2 to static analysis of JavaScript. Big CFA2 is the part of DoctorJS that is not specific to JavaScript.

DoctorJS was built in collaboration with Mozilla. It is open source and available from `github.com/mozilla/doctorjs`.

7.1 Basics of the JavaScript object model

Object properties

A JavaScript object is a map from property names to values. A property name is a string. Unlike Java, the number of properties in an object can change dynamically; at runtime, properties can be added to and deleted from objects at will. If we assign to a non-existent property, the property gets created automatically.

There are two ways to access a property of an object.

- Dot notation: In the expression `obj.foo`, `obj` is an arbitrary expression that evaluates to an object and `foo` is a string constant, which acts as the property name.
- Computed property access: In the expression `obj[foo]`, `foo` is not a constant, it is an arbitrary expression. If `foo` evaluates to a value v that is a string, v becomes the property name. If not, v is coerced to a string, which becomes the property name.

This semantics has two important implications. First, since an abstract object must have a bounded number of properties, some properties of a concrete object must be merged together in the analysis. Second, in the expression `obj[foo]`, finding which string `foo` evaluates to is undecidable. Therefore, field-sensitive analysis [55] is undecidable for JavaScript.

Prototypal inheritance

JavaScript does not have classes. Inheritance is object-based. When an object o is created, it is assigned a prototype object. When we try to access a property p of o, if o has the property, its value is returned. Otherwise, we look for p in o's prototype, and in the prototype of o's prototype and so on, until either p is found or we reach the top of the prototype chain. If p is not found, the special value `undefined` is returned. Every object has at most one prototype and the runtime enforces that the prototype chain has no cycles.

Object creation

There are two ways to create an object. The first is by using an object-literal expression. For example, the expression `{a: 5, b: "hello"}` evaluates to an object with properties `a` and `b`. Objects created this way are assigned some default prototype.

Alternatively, one can call a function with the `new` keyword. For example, if we have the following function

```
function Foo(x) {
  this.a = x + 5;
  this.b = "hello";
}
```

we can call `new Foo(13)` to create an object with two properties. A function can be called both with and without `new` in the same program, but in practice this rarely happens. By convention, when a function is used to create objects, its name starts with a capital letter, it is always called with `new` and people refer to it as a *constructor*. Every object created by a call `new Foo(...)` is said to belong in "class" `Foo`.

As we mentioned, functions are objects. By default, every function object has a `prototype` property, which points to some object o; o is *not* the prototype of the function. We can control inheritance by assigning to the `prototype` property. If we say

```
Foo.prototype = {c: 94};
```

then `{c: 94}` becomes the prototype of all subsequently created `Foo` objects.

7.2 Handling of JavaScript constructs

Abstracting objects affects inheritance

In a static analysis, some properties of a language that are intuitive and well established suddenly cease to hold because of approximation. For example, a sound static analysis for JavaScript must allow cycles in the prototype chain and objects that have more than one prototype. These issues arise regardless of the abstraction used. (Naturally, a more precise abstraction runs into these issues less often.)

(a) Concrete (b) Abstract

Figure 7.1: Example: cycle in the prototype chain

The following snippet shows how an abstract object can have many prototypes. The abstract value of the expression `exp` becomes the prototype of `o1`. For any given abstraction, we can cause `exp` to evaluate to a set of two or more abstract objects, which are all possible prototypes of `o1`.

```
function Foo() = { ... };
Foo.prototype = exp;
var o1 = new Foo();
```

A program can create an unbounded number of objects. Therefore, some concrete objects are approximated by the same abstract object during static analysis. Approximation can create cycles in the prototype chain.

```
1  function create(Constructor) {
2    return new Constructor();
3  }
4  function Parent() { ... }
5  function Kid() { ... }
6  Kid.prototype = create(Parent);
7  var k = create(Kid);
```

In this program, the higher-order function `create` takes a constructor as an argument and returns an object. The first call to `create` returns a `Parent` object `p`. Since we have not changed the default `prototype` property of `Parent`, `p` is assigned a default prototype object `d`. The second call to `create` returns a `Kid` object `k`, whose prototype is `p` (fig. 7.1a).

A common abstraction for objects is to merge all concrete objects allocated at the same program point to a single abstract object. With this abstraction, the analysis merges the `Parent` and `Kid` objects to a single abstract object created at line 2. A possible prototype of this object is itself (fig. 7.1b).

DoctorJS allows cycles in the prototype chain and objects with many prototypes. When we do a property lookup, we record seen objects as we go up the chain to detect cycles and ensure termination.

Arrays used as tuples

JavaScript does not have tuples. Instead, programmers often use array objects as tuples. The elements of a tuple can be heterogeneous, so we try to distinguish them to avoid merging.

```
1  var a, i, x, y, z;
2  a = [1, "abc"];
3  x = a[0];
4  y = a[i];
5  z = a[0];
```

In this program, we create an array with two elements at line 2. At line 3, we find that `x` is bound to a number. At line 4, the index is a variable, so we merge the properties "0" and "1" to a single property that is bound to either a number or a string. At line 5, we find that `z` is bound to either a number or string.

In general, we keep the properties of an array separate until the first time it is accessed with a non-constant index. Then, we assume that the array is not a tuple and merge its elements. Every subsequent access gets the join of all elements.

Technically, all property names in JavaScript are strings. However, the string "31" is numeric, while "hello" is not. DoctorJS distinguishes between numeric and non-numeric property names to increase precision. For instance, we may merge all elements of an array of strings together, but we do not merge them with the "length" property, so we can find that the value of "length" is a number.

Out-of-bounds array indices

Consider the expression `a[i]` where `a` evaluates to an array and `i` to a number. If `i` is a valid index in the array, the corresponding element is returned. If not, the result is the `undefined` value—no exception is thrown. (Accessing an array element is the same as any other property access.)

(a) Concrete (b) Abstract

Figure 7.2: Example: property assignment

The possible analysis choices for `a[i]` are a good demonstration of the trade-off between speed, precision and soundness.

- *Fast, imprecise, sound*: If all numbers abstract to a single abstract number $\mathbb{R}$, then the analysis cannot know if the access is in bounds. For soundness, every element access must include `undefined` in the resulting abstract value. Clearly, this is imprecise.
- *Fast, precise, unsound*: If all numbers abstract to $\mathbb{R}$, we can be precise by optimistically assuming that most accesses will be in bounds. Then, we return the join of the array elements, but not `undefined`.
- *Slow, precise, sound*: To analyze `a[i]` both safely and precisely, one must use a more complex lattice for abstract numbers that can reason about numeric ranges. In this case, lattice operations are slower.

DoctorJS currently uses the second solution.

Assigning to a property

To evaluate an expression such as `obj.x = 21`, we first evaluate `obj` to an object o. If o has a property named `x` that is *not inherited*, we set the value of `x` to 21. Otherwise, *we create a new property* named `x` on o with value 21. The new property may shadow an inherited property with the same name.

New property Adding properties to objects during static analysis is not straightforward. Suppose there are two objects `o2` and `o3` which inherit from `o1` (fig. 7.2a); `o1` has a property `x` bound to a string. Now let's assume that a static analysis cannot distinguish `o2` and `o3`; they both abstract to an object `ao2`, and `o1` abstracts to `ao1` (fig. 7.2b).

How would we analyze `obj.x = 21`, where `obj` evaluates to `ao2`? The first solution that comes to mind is to add the property `x` to `ao2`, which is unsound. In the concrete, it is possible that the assignment happens only for

one of the two objects, say `o3`. Then, `o3.x` is 21 and `o2.x` is still `"foo"`. If we add `x` to `ao2`, then we shadow `ao1.x`, so `ao2.x` cannot be a string.

A sound analysis cannot unconditionally add a property to an object. It must mark the new property as "possibly there." If we look up such a property, we have to keep searching up the prototype chain for properties with the same name. If all occurrences of a property in the chain are "possibly there," we must include `undefined` in the result.

The sound solution is imprecise in the case when a new property is added to all concrete objects approximated by an abstract object. Also, it slows down property lookups. DoctorJS adds new properties to abstract objects unconditionally, which is unsound.

Existing property When assigning to an existing property, we just join the rvalue with the value of the property. However, the assigned property may be a computed property, as in `obj[exp] = 21`. In DoctorJS, if we cannot find which string `exp` evaluates to, we merge all properties of the object into one and do the assignment.

Because of computed properties, the choice of abstraction for strings has a big impact on a JavaScript static analysis. Approximating all strings with a single abstract string is fast but imprecise; computed properties are very common. In DoctorJS, an abstract string is either a string literal that appears in the program text or an unknown string. So, we propagate string constants, but any operation on strings immediately goes to top. Even though this abstraction is more precise than a single abstract string, we would like to try more sophisticated lattices to improve precision for computed properties.

Property deletion

Deleting a property from an object raises similar issues for an analysis as creating a property by assignment. Since an abstract object generally represents many concrete objects, deleting the property is unsound, because in the concrete semantics the deletion may happen only to some of the objects. The sound and slower solution is to mark the property as "possibly there."

Our solution in DoctorJS is to do nothing; we leave the abstract object unchanged. This is the simpler of the two unsound solutions. Property deletion is rare in practice [53], so handling it unsoundly does not change the analysis results substantially.

The `prototype` property of constructors

For every function f in the program, JavaScript automatically creates an object o and sets the `prototype` property of f to o. If f is used as a constructor, o becomes the prototype of the new object. The default object o is rarely useful, so programmers commonly set the `prototype` property of constructors to an object o' of their choosing.

We want to avoid binding the `prototype` property to a set $\{o, o'\}$, if only o' is going to be used. For this reason, we create the `prototype` property on functions *lazily*. During the analysis, if a function f is called with `new`, we check if it has a `prototype` property. If so, we use the referenced object as the prototype of the new object. If not, we create the property and a default object at that moment. This way, when programmers replace the default object o with o', the prototype of objects created by f is $\{o'\}$, not $\{o, o'\}$.

Core objects

JavaScript provides a small set of objects in every host environment, even in a non-browser environment. Some of these are `Object`, `Function`, `Array`, `Number`, `String` and `RegExp`. DoctorJS can reason about the core objects.

At the beginning of the analysis, we set up the abstract heap. We first create the *global object*. All globally available names, such as `NaN` and `Infinity` become properties of the global object. We create one abstract object in the heap for every core object and add properties to the global object that point to the core objects, *e.g.*, the global variable `Array` points to the abstract `Array` object.

Some core methods can be called on scalars. For example, the expression `"abc".charAt(0)` is legal; `"abc"` is coerced to a `String` object, which becomes the target object of the method call. We model this behavior by creating dummy `Number`, `String` and `Boolean` objects during the initialization phase. When a method is called on a scalar, we use the dummy object of the same type for the call.

CHAPTER 8

Evaluation

This chapter presents results from two implementations of our analyses. In section 8.1, we discuss a Scheme implementation of CFA2 without first-class control. In section 8.2, we discuss an implementation of Big CFA2 for JavaScript. Our experimental results provide evidence to support the thesis that pushdown models are more suitable than finite-state models for higher-order flow analysis.

8.1 Scheme

To support our claim that call/return matching improves dataflow information, we used CFA2, 0CFA and 1CFA to perform constant propagation and folding. We implemented the three analyses in the Twobit compiler [10], for a subset of Scheme with numbers, booleans, pairs and explicit recursion.

We implemented CFA2 with stack filtering. Also, we did not widen the targets of path edges to prevent tail duplication (see sec. 4.6.1). We only widened the abstract heap; instead of a heap per state, we used a global heap and states carry timestamps. Similarly, we used a global variable environment and timestamps on states for 0CFA and 1CFA.

We compared the effectiveness of the analyses on a small set of benchmarks (table 8.1). We measured the number of stack and heap references in each program and the number of constants found by each analysis. The implementations are unoptimized prototypes, so we did not measure running times. To estimate how well an optimized implementation would do, we counted the number of elements inserted in the workset by each analysis. (Note that 0CFA and 1CFA visit abstract states, whereas CFA2 visits pairs of

	LOC	$S_?$	$H_?$	0CFA		1CFA		CFA2	
				visited	const.	visited	const.	visited	const.
`len`	6	9	0	81	0	126	0	55	2
`rev-iter`	6	17	0	121	0	198	0	82	4
`tree-count`	6	33	0	293	2	2856	6	183	10
`len-Y`	11	15	4	199	0	356	0	131	2
`flatten`	13	37	0	1520	0	6865	0	478	5
`ins-sort`	14	33	5	509	0	1597	0	600	4
`church-nums`	35	46	23	19130	0	19411	0	22671	0
`sets`	41	90	3	3915	0	54414	0	4251	4
`DFS`	42	94	11	1337	8	6890	8	1719	16

Table 8.1: Scheme mini-benchmark results

states.) The running time of an analysis increases as the size of the state space increases.

The chosen benchmarks exhibit a variety of control-flow patterns. `Len` computes the length of a list recursively. `Rev-iter` reverses a list tail-recursively. `Tree-count` counts the nodes in a binary tree. `Len-Y` computes the length of a list using the Y-combinator instead of explicit recursion. `Flatten` turns arbitrarily nested lists into a flat list. `Ins-sort` sorts a list of numbers using insertion-sort. `Church-nums` tests distributivity of multiplication over addition for a few Church numerals. `Sets` defines the basic set operations and tests De Morgan's laws on sets of numbers. `DFS` does depth-first search of a graph.

CFA2 finds the most constants, followed by 1CFA. 0CFA is the least precise. Also, the increased precision of CFA2 creates fewer spurious flows, so the state space is small. Even though we did not do much widening for CFA2, its visited set is comparable in size to that of 0CFA. In five out of nine cases, CFA2 visits fewer paths than 0CFA does states. The visited set of CFA2 can be up to 3.2 times smaller (`flatten`), and up to 1.3 times larger (`DFS`) than the visited set of 0CFA. The state space of 1CFA is larger than both 0CFA (9/9 cases) and CFA2 (8/9 cases). Moreover, the state space of 1CFA can be significantly larger than that of CFA2 in some cases (15.6 times in `tree-count`, 14.4 times in `flatten`, 12.8 times in `sets`).

Naturally, the number of stack references in a program is much higher than the number of heap references; most of the time, a variable is referenced only by the lambda that binds it. Thus, CFA2 uses the precise stack lookups more often than the imprecise heap lookups.

8.2 JavaScript

In chapter 7, we presented DoctorJS, a static analysis for JavaScript based on Big CFA2. DoctorJS is itself written in JavaScript, using the NodeJS framework.[1] We use Narcissus for lexing and parsing.[2] After parsing, we perform several transformations on the AST before static analysis, *e.g.*, name variables uniquely, desugar away some constructs, classify references into stack and heap.

DoctorJS is not tied to a particular analysis client; it can potentially be used for several kinds of tools. So far, we have used DoctorJS for type inference, and to search how often specific program behaviors happen in a large code base. In this section, we discuss these two applications and report on some experiments we conducted. For the experiments, we used a machine with an Intel E8400 processor (2 cores, 3GHz, 6MB L2 cache), 2GB of RAM, running Ubuntu 11.10.

8.2.1 Type inference

A common and useful application of flow analysis in a dynamic language is type inference. Type information can be used to find bugs, or in an IDE to provide informative code completion, help with refactoring, *etc*. We wrote a simple type-inference pass for DoctorJS. After the analysis, we process the results and turn abstract values into types, which can be presented to the user or written to a file in some standard format.

An abstract value is a (possibly empty) set of abstract scalars and objects. Numbers abstract to a single abstract number. An abstract boolean is `true`, `false` or an unknown boolean. An abstract string is either a string literal that appears in the program text or an unknown string. `Null` and `undefined` abstract to themselves. All concrete objects created at the same program point are represented by a single abstract object that maps a finite number of property names to abstract values.

We turn an abstract value, which is a set S, into a type by taking the union of the types of the elements of S. Abstract strings have type `string` and abstract booleans have type `boolean`. The other abstract scalars can be used as types without change. The type of an abstract object depends on

[1] `nodejs.org`
[2] `github.com/mozilla/narcissus`

whether the object is a function, an array, or some other object.

The type of a function describes the types of the arguments and the result. We do not use type variables in the types of polymorphic functions. In the following program, `id` is called with a number and a string.

```
function id(x) { return x; }
var n = id(8);
var s = id("hello");
```

The inferred type for `id` is $\{\texttt{number}, \texttt{string}\} \rightarrow \{\texttt{number}, \texttt{string}\}$. This type is not ideal; it implies that `id` could take a number and return a string. This imprecision does not impact the call sites of `id`; because of call/return matching, we find the precise types for `n` and `s`.

In order to calculate the input and output types of a function, we may have to traverse the object graph in the abstract heap. If we run into circular objects, we return the type `any`. For example, the inferred type for `id` in the following program is $\texttt{any} \rightarrow \texttt{any}$.

```
function id(x) { return x; }
id(id);
```

The type of a heterogeneous array is `Array`. For a homogeneous array, we also report the type of the elements. For other objects, we use the name of the constructor as the type.

Benchmarks We ran the type inference on two well-known benchmark sets, the V8 benchmarks (version 6) and the SunSpider benchmarks. The results appear in table 8.2. We measured the running time and memory requirements of the analysis (median values for 3 runs, no significant variation).

The results show that the analysis is fast and lightweight. Each SunSpider benchmark is a few hundred lines or less. (Regexp-dna is 1708 lines because of a big input string, the actual code is less than 100 lines.) The median analysis time is 110ms and the median memory consumption is 5.8MB. The V8 benchmarks range from 394 to 4684 lines of code, with a median analysis time of 648ms and a median memory consumption of 14.2MB.

After manual inspection of the results, we found that the inferred types are usually precise. For example, table 8.3 shows the types of the top-level functions in SunSpider's crypto-aes program. We write `[num]` to mean an

SunSpider	size (KB)	size (LOC)	time (ms)	mem (MB)
3d-cube	8.5	341	332	8.4
3d-morph	2	34	100	5.2
3d-raytrace	34.5	418	335	10.4
access-binary-trees	1.2	48	106	5.8
access-fannkuch	1.5	62	110	6.1
access-nbody	4	167	138	6.8
access-nsieve	0.8	35	103	5.1
bitops-3bit-bits-in-byte	0.8	20	94	4.6
bitops-bits-in-byte	0.4	20	93	4.5
bitops-bitwise-and	1.5	4	90	4.1
bitops-nsieve-bits	0.8	35	100	5.2
controlflow-recursive	0.6	23	101	5
crypto-aes	16.7	412	268	11.8
crypto-md5	9.6	274	216	9.2
crypto-sha1	6.3	212	147	7.9
date-format-tofte	32.6	285	176	6.2
date-format-xparb	12	410	195	6.6
math-cordic	2.7	74	103	5.5
math-partial-sums	1.8	36	108	6
math-spectral-norm	1	48	105	5.8
mont	3.3	117	126	6.6
regexp-dna	105.7	1708	103	5.1
string-fasta	2	85	113	6.3
string-tagcloud	8.4	233	153	8.4
string-unpack-code	163.2	60	141	4.4

V8	size (KB)	size (LOC)	time (ms)	mem (MB)
crypto	46.7	1698	4710	24.1
deltablue	25	880	648	13
earley-boyer	190.1	4684	3735	36.8
raytrace	27.3	904	2923	14.2
raytrace inline	27.7	904	467	14.8
regexp	108.6	1761	838	19.1
richards	15.3	539	190	6.8
splay	10.4	394	163	4.7

Table 8.2: Type-inference performance results

Function name	Inferred type
Cipher	[num], [[num]] → [num]
SubBytes	[[num]], num → [[num]]
ShiftRows	[[num]], num → [[num]]
MixColumns	[[num]], num → [[num]]
AddRoundKey	[[num]], [[num]], num, num → [[num]]
KeyExpansion	[num] → [[num]]
SubWord	[num] → [num]
RotWord	[num] → [num]
AESEncryptCtr	str, str, num → str
AESDecryptCtr	str, str, num → str
escCtrlChars	str → str
unescCtrlChars	str → str
encodeBase64	any → str
decodeBase64	any → str
encodeUTF8	any → any
decodeUTF8	str → str
byteArrayToHexStr	any → str

Table 8.3: Types for the top-level functions in crypto-aes

array of numbers and `[[num]]` for an array of arrays of numbers. For the functions up to `unescCtrlChars`, we find the exact type. The remaining functions are dead code, so the analysis assumes that their arguments can have any type. Even so, it is able to pick up their return types correctly. For `DecodeUTF8` we can also find the input type, because it is called with a string from `decodeBase64`.

Last, it is worth mentioning how the choice of abstraction for objects impacts the analysis. DoctorJS currently does no heap specialization, *i.e.*, all concrete objects allocated at the same program point are approximated by one abstract object. In the V8 raytrace benchmark, the creation of many different objects goes through the same wrapper:

```
var Class = {
  create: function() {
    return function() {
      this.initialize.apply(this, arguments);
    }
  }
}
```

There is a call to `Class.create` in 14 program points. The call returns a function object which acts as a constructor, but delegates all the work to the appropriate `initialize` function. Since we model objects without context in the heap, we create one abstract object for the returned function. As a result, many different places modify the `prototype` property of the same function and we get spurious flows. With one level of context sensitivity for objects, we would have 14 different functions that do not get conflated. In fact, when we inlined the result of the call in the 14 sites and reran the analysis, there was a 6.3× speed-up (see raytrace inline).

8.2.2 Analysis of Firefox add-ons

Besides type inference, we also used DoctorJS to analyze all Firefox add-ons, as part of the Electrolysis project at Mozilla. Electrolysis was a proposal for changing the architecture of Firefox.[3] Among other things, Electrolysis changes how a Firefox add-on and a web page interact; an add-on can only interact with the web page through a restricted API.

When considering the pros and cons of Electrolysis, Mozilla wanted to know how often add-ons interact with a web page in ways that would no longer be possible with Electrolysis. Each such interaction is code that would have to be rewritten. One way to estimate the extent of the rewrite is with a static analysis. We used DoctorJS to analyze all add-ons and look for patterns that would be forbidden with Electrolysis.

Chrome and content elements In a browser environment, objects are specified using the Document Object Model (DOM). At runtime, objects form a tree called the DOM tree.

The HTML elements of a web page are a subtree of the DOM tree. Most web pages also have JavaScript code that manipulates the elements. These elements and the associated JavaScript are loaded dynamically from the web when a user visits a web page. We refer to the objects and the JavaScript code that are part of a web page as *content*.

Part of the Firefox user interface also consists of DOM objects, which are manipulated using JavaScript. The code of an add-on is written in JavaScript and can interact with the browser UI elements and the objects of a web page. We refer to the objects in the browser UI and the JavaScript code of

[3] wiki.mozilla.org/Electrolysis

the browser and the add-on as *chrome*. Unlike content code, chrome code is installed locally on the user's machine.

With the current Firefox architecture, chrome and content interact freely. We are interested in two patterns that are not possible with Electrolysis.

- References from chrome to content: Currently, a property of a chrome object can point to a content object. With Electrolysis, chrome objects cannot directly reference content objects.
- Listening for content events: When some interesting change in the state of the system happens, such as a key press or a mouse click, the browser creates an *event*. A *listener* is a function that is attached to an element on the DOM tree in order to handle an event. An event may have a target element, *e.g.*, a user clicks on a specific part of a page. If there is no listener attached to the target element, the event may bubble up the DOM tree until a listener is found.[4] Currently, a content-generated event can bubble up past the root of the content subtree and be handled by a listener attached to a chrome element. With Electrolysis, a listener on a chrome element cannot listen for content events, and chrome code cannot attach listeners to content elements.

The details of these decisions will not concern us here. The rationale is that the separation between chrome and content could increase performance on multicore machines and also increase protection from malicious web pages.

Searching for violations of the Electrolysis API in Firefox add-ons We set out to analyze all add-ons and find occurrences of the two aforementioned patterns in the code. There are two obstacles in doing this analysis. First, the DOM specification is huge, so a complete modeling of the DOM in the analysis was out of the question for us. Second, we cannot have a full picture of the execution. In reality, the add-on is installed on a user's browser and the user loads arbitrary web pages and interacts with them, causing events to fire. Instead, we have the add-on code but we do not know anything about the web pages or the user's actions. Both these difficulties can be overcome.

We created a small DOM mockup that is expressive enough for our pur-

[4]Not all events bubble. For more details, see `en.wikipedia.org/wiki/DOM_events`.

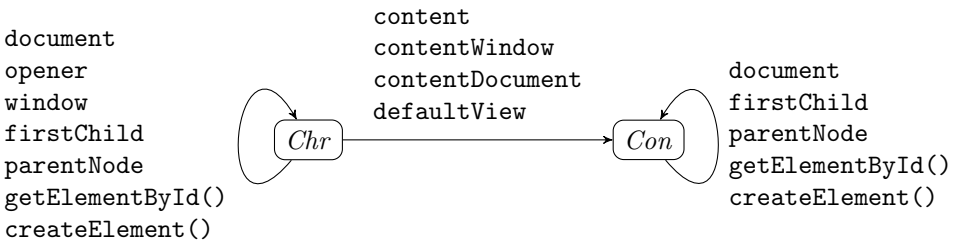


Figure 8.1: DOM mockup

poses. We approximate all chrome objects by a single object Chr and all content objects by a single object Con.

We cataloged the most commonly used properties and methods and added them to Chr and Con. Fig. 8.1 shows some of the properties and methods we model. In the browser, the global object is `window`, which is a chrome object, so Chr is the global object in the analysis. Hence, the global variables are properties of Chr. For example, Chr has properties `window` and `opener` that point back to Chr and properties `content` and `contentWindow` that point to Con.

We added about a dozen well-known properties and methods to Chr and Con; there are numerous properties that we do not model. Suppose that during the analysis we find an expression `obj.p` and `obj` evaluates to either Chr or Con and does not have a property named `p`. It is possible that `p` does not exist and we should return `undefined`. But most likely, `p` is not there because it is one of the properties that we do not model. We assume the latter and return whatever `obj` evaluates to as the result of the expression. This simple unsound heuristic is crucial because it allows us to ignore a large number of properties and still get useful results from the analysis; we only need to know about the small number of properties that can take us from Chr to Con.

During the analysis, when we find a property access where the target object can evaluate to Chr and the referenced property can point to Con, we mark it as a violation.

The analysis does not simulate the firing of events. When we see a listener l being attached to some object, we assume that the event for which l listens will fire. Attaching a listener happens by calling `addEventListener`. A call to this function has the general form:

```
obj.addEventListener(evttype, listener, capture, untrusted)
```

where `obj` is the element to which the listener will be attached, `evttype` is a string that is the name of the event and `listener` is the function that handles the event. (To avoid getting sidetracked into the details of the event model, we do not discuss `capture` and `untrusted` here.)

We split events into three categories: XUL events (specific to the Firefox UI), non-XUL events that bubble and non-XUL events that do not bubble. The name of an event determines which category the event belongs to. XUL events cannot be generated from content. The other two kinds of events can be generated both from chrome and content.

During the analysis, every time we see a call to `addEventListener`, we use the following criteria to decide whether to flag it as a violation of Electrolysis. If `evttype` is a XUL event, the call is safe. Otherwise, we look at `obj`. If `obj` can evaluate to Con, the code of the add-on is attaching a listener to a content object, so we mark this as unsafe. If `obj` evaluates to Chr and `evttype` is an event that does not bubble, the call is safe. Last, if `obj` evaluates to Chr and `evttype` is an event that bubbles then the call is unsafe, because we are attaching a listener to chrome that may handle an event that originated from content.

Benchmark results A Firefox add-on consists of several files that contain code, user interface markup, documentation, *etc*. The add-on is compressed as an `xpi` file and distributed. David Herman of Mozilla wrote a program to unpack an `xpi` and collect the JavaScript code from all files into a single file. This file is the input to DoctorJS.

We analyzed 6997 add-ons and found a total of 40138 violations of the Electrolysis API, an average of about 6 violations per add-on. From these results, we concluded that Electrolysis requires extensive rewrites to add-ons code. For this and other reasons, Mozilla decided to not go through with Electrolysis.

Table 8.4 shows some aggregate statistics. We group the add-ons into 13 categories depending on their size in kilobytes. Many add-ons use JavaScript libraries such as jQuery that are minified in the source code and line breaks are removed. Therefore, lines of code are not a precise indicator of size.

The add-ons vary widely in size, but most add-ons are small. We set a timeout of 5 minutes and if the analysis does not finish by that time, we stop

size (KB)	#add-ons	#timeouts	time (ms)	mem (MB)	viol
2	1064	0	7	1.2	0
4	873	0	22	2.1	0
8	1063	0	44	1.8	0
16	1092	0	93	3.7	1
32	818	0	189	6.7	2
64	717	66	414	14.2	4
128	485	120	947	20.5	5
256	335	116	2122	27.9	9
512	427	301	9233	34.6	4
1024	93	69	8231	77.2	8
2048	24	7	68035	249.4	57
4096	4	4	—	435.8	20
8192	2	2	—	592.4	3

Table 8.4: Aggregate results for all add-ons

	$<$1s	$<$2s	$<$3s	$<$4s	$<$5s	$<$10s	$<$20s	$<$60s	$\geqslant$60s	∞
128KB	195	118	21	9	5	8	4	2	3	120
256KB	12	89	50	20	13	15	9	6	5	116

Table 8.5: Distribution of running times for the 128KB and 256KB add-ons

it. The analysis times out for 685 add-ons (9.8%). When the analysis times out for an add-on, we record the violations found up to that point.

We report how many add-ons fall in each size category. For each category, we report the number of add-ons that time out, the median running time for the add-ons that do not time out, the median memory consumption and the median number of violations.

For add-ons in the same size category, the running time of the analysis varies a lot. Table 8.5 shows a distribution of the running times for add-ons of size 128KB and 256KB. Most 128KB add-ons finish in under a second, but 3 add-ons take more than one minute. The majority of the 256KB add-ons finish in under 3 seconds, but 5 add-ons take more than a minute.

Table 8.6 shows the analysis results for some popular add-ons (median values for 3 runs, no significant variation). We see that DoctorJS analyzes some large add-ons such as Firebug quickly and without using much memory. We manually inspected the reported violations for these add-ons to see if the analysis gives accurate results. Of the 340 reported violations, 303 are true violations and 37 are false positives.

	KB	LOC	time (ms)	mem (MB)	viol.
Commentblocker	26	762	135	5.5	23
Flashblock	33	936	183	8.7	2
Imtranslator	49	1335	293	3.4	6
Flagfox	92	2056	556	5.1	5
Greasemonkey	193	5764	1103	23.9	10
Flashgot	316	9730	5323	24.5	12
Video download helper	409	12916	2997	57.6	6
Web developer	812	20493	5993	33.6	86
FoxyTunes	1179	35200	8401	109.8	49
ReminderFox	1296	35980	194050	72.8	1
Firebug	2598	80480	30696	271.8	140

Table 8.6: Analysis results for some Firefox add-ons

Overall, our experiments with DoctorJS support the ideas in Big CFA2. Call/return matching and the stack/heap distinction reduce spurious flows, as indicated by the low number of false positives. By not caching intermediate results, the analysis uses little memory.

Interestingly, after examining some of the add-ons that time out, we found that the main remaining cause of slowness in DoctorJS is unrelated to the pushdown abstraction; it is related to the abstract heap. We represent the heap as a *finite map from addresses to abstract values*. This abstraction works well most of the time, but it can be problematic when several addresses point to the same abstract value. This happens often if the heap of the analyzed program is highly connected. The following contrived example illustrates this phenomenon.

```
function f(u) {
  w = u;
  x = w;
  y = x;
  z = y;
}
```

We have heap references to `w`, `x`, `y`, `z`. Initially, these variables are bound to $\emptyset$ in the heap. Suppose that `f` can be called with n different abstract objects. These objects will flow through `u` to all four variables. Every time the heap binding of one of these variables grows, the timestamp increases. Thus, the

timestamp increases $4n$ times total. Every timestamp increase creates work for the analysis because it deprecates summaries and causes functions to be reanalyzed.

Instead of $4n$ increases, we only need n, one for each abstract object that flows to `w`, because it is guaranteed to also flow to `x`, `y` and `z`. But because we represent the heap as a table from addresses to values, the bindings `heap[w]`, `heap[x]`, `heap[y]` and `heap[z]` are totally independent; we do not see the invariant

$$\texttt{heap[w]} \subseteq \texttt{heap[x]} \subseteq \texttt{heap[y]} \subseteq \texttt{heap[z]}$$

If we represent the heap as a *graph of points-to relations*, like constraint-based analyses do, we can express this invariant and increase the timestamp only n times.

CHAPTER 9

Related work

9.1 Program analyses for first-order languages

The earliest pushdown program analysis for first-order languages is Sharir and Pnueli's *functional approach* [59]. In its general form, the analysis has two stages. First, they compute a transfer function for each procedure f by composing the transfer functions of its basic blocks. The second stage is an ordinary fixpoint computation over the control-flow graph of the whole program; at every call site of f, they use the previously computed transfer function to simulate the effect of the call. The major difficulty with the functional approach is how to represent the transfer functions when the dataflow lattice is infinite. In this case, a table representation is impossible and there is no general way of finding when a function admits a symbolic representation. In the special case when the lattice is finite, they give an *iterative* algorithm, which is the first summary-based analysis (pg. 207).

The functional approach, as proposed by Sharir and Pnueli, works for procedures without arguments. Knoop and Steffen extended it for procedures that take arguments and have local-variable declarations [35]. They use a stack to analyze different calls in different environments. Like CFA2, they give a terminating algorithm that remembers only a finite part of the stack without losing precision. Their algorithm uses the two-stage approach of Sharir and Pnueli, not the one-stage iterative summarization. Since their work is for first-order languages, they do not have to deal with closures and do not make a distinction between stack and heap references.

Reps *et al.* [52] showed how to perform pushdown analysis efficiently for a certain class of dataflow problems. When the dataflow facts are sets and the transfer functions are distributive, the transfer functions admit a

compact representation and the analysis problem can be turned into a graph reachability problem. Their *tabulation algorithm* solves the reachability problem without merging any information from distinct call sites of the same procedure. The CFA2 algorithm (fig. 4.8) uses the terminology of Reps *et al.*, but its mechanics are closer to Sharir and Pnueli's summarization.

Debray and Proebsting [16] used ideas from parsing theory to design an interprocedural analysis for first-order programs with tail calls. They describe control-flow with a context-free grammar. In the monovariant version of their analysis, the FOLLOW set of a procedure represents its possible return points. The polyvariant version uses one level of context, like 1CFA. In this version, the LR(1) items show return-point information. The authors do not discuss how to handle tail calls in an analysis with unbounded call/return matching.

9.2 Polyvariant analyses for object-oriented languages

In this section, we discuss analyses for object-oriented languages except JavaScript. We discuss JavaScript analyses in section 9.5.

There is little work on pushdown analyses for object-oriented languages. Most work has focused on finite-state analyses for Java. The Paddle and Doop frameworks represent the state of the art in Java points-to analysis.

The Paddle framework of Lhoták and Hendren uses BDDs to represent analysis results compactly [40]. The authors used Paddle for an extensive comparison of different options for polyvariance. They found that object sensitivity [44] is a better choice of context than call strings and that the 1-object-sensitive analysis with heap specialization (*abbrev.* 1H-obj) offers the best trade-off between precision and scalability. Using 1H-obj, they were able to analyze programs up to 131KLOC, with running times ranging from a few minutes to a half hour. They also found that increasing the call-string size does not increase precision much, but greatly increases the state space, which is consistent with our Scheme experiments (sec. 8.1).

The Doop framework of Bravenboer and Smaragdakis allows declarative specification of analyses using Datalog [7]. The authors suggest an optimization methodology for Datalog, which achieves order-of-magnitude speed-ups

over previous approaches. Bravenboer and Smaragdakis compared Doop to Paddle on the DaCapo benchmarks, the largest of which is 131KLOC [40]. Interestingly, they state that these benchmarks are the largest programs anyone has ever applied a polyvariant points-to analysis on. They found Doop to be up to 15 times faster than Paddle. On the other hand, Paddle represents large dataflow relations with a lot of redundancy very compactly, in a few megabytes, whereas Doop may need gigabytes for the same relations. The experiments with Doop confirmed Lhoták and Hendren's finding that object sensitivity is a preferable abstraction to call strings.

Sridharan and Bodík's demand-driven analysis for Java matches reads with writes to object fields selectively, by using refinement [66]. The idea is to start with an approximate but sound solution and make it more precise in parts of the program that matter by using pushdown techniques. The analysis also refines calls and returns, but approximates in the presence of recursion. In their experiments, their analysis was able to prove the safety of 61% more casts than 1H-obj.

Agesen developed the Cartesian Product Algorithm (CPA) for type inference of Self [1]. CPA analyzes each function once for every combination of abstract values that can flow to the formal parameters. Thus, CPA uses argument tuples as an abstraction of context. By using summaries, CFA2 also distinguishes calls that happen with different arguments. However, CFA2 uses sets as abstract values and does not compute the Cartesian product. It would be interesting to combine CPA and CFA2 in a pushdown analysis that creates one summary for every element of the Cartesian product.

We believe that Java and object-oriented languages in general would benefit from a pushdown analysis like CFA2. First, CFA2 distinguishes calls to the same function with different arguments. This generalizes object sensitivity, which distinguishes calls depending only on the 0^{th} argument, *i.e.*, the receiver object. Second, pushdown analyses capture the control flow induced by calls and returns more naturally than finite-state analyses, especially in the presence of recursion. Past comparisons have found pushdown analyses to be more precise than finite-state analyses [52, 66]. Our Scheme experiments support this finding (sec. 8.1). Last, our experiments with DoctorJS show that the increased precision of CFA2 is attainable in practice: DoctorJS scales decently to large programs, even with our suboptimal representation of the abstract heap (sec. 8.2.2).

9.3 Polyvariant analyses for functional languages

In his dissertation, Might proposed finite-state analyses to solve the environment problem [42]. These analyses are based on a reformulation of kCFA using the operational semantics of a CPS λ-calculus. We find the operational version of kCFA convenient to work with and simpler than Shivers's original denotational formulation [61]. This is why we also use CPS and operational semantics for CFA2. Might's first solution to the environment problem, μCFA, counts how many times a variable is bound during the analysis in order to show equality of environments. The second solution, ΔCFA, compares the environments of abstract states by looking at stack changes between those states. ΔCFA uses "frame strings" to track stack motion. Frame strings can describe various sorts of control transfer: recursive call, tail call, return, primitive application and exotic stack changes caused by first-class control operators. However, the expressiveness of ΔCFA is hampered by using a finite-state abstraction. It is possible that frame strings could be combined with a pushdown abstraction to boost precision of ΔCFA-based analyses.

Mossin [45] created the first analysis with unbounded call/return matching for *typed* functional languages. It is a type-based flow analysis, which uses polymorphic subtyping for polyvariance. The input to the analysis is a program p in the simply-typed λ-calculus with recursion. First, the analysis annotates the types in p with labels. Then, it computes flow information by assigning labeled types to each expression in p. Thus, flow analysis is reduced to a type-inference problem. The annotated type system uses let-polymorphism. As a result, it can distinguish flows to different references of let- and letrec-bound variables. In the following program, it finds that `n2` is a constant.

```
(let* ((id (λ (x) x))
       (n1 (id 1))
       (n2 (id 2)))
  (+ n1 n2))
```

However, the type system merges flows to different references of λ-bound variables. For instance, it cannot find that `n2` is a constant in the `app/id` example (see fig. 3.1). Mossin's algorithm runs in time $O(n^8)$.

Rehof and Fähndrich [50, 19] used CFL reachability in an analysis that runs in cubic time and has the same precision as Mossin's. They also extended the analysis to handle polymorphism in the target language. Around the same time, Gustavsson and Svenningsson [26] formulated a cubic version of Mossin's analysis without using CFL reachability. Their work does not deal with polymorphism in the target language.

Recently, Earl *et al.* created PDCFA, a pushdown analysis that does not use frames [17]. Rather, it allocates all bindings in the heap with context, like kCFA. For $k = 0$, PDCFA runs in time $O(n^6)$. In CFA2, it is possible to disable the use of frames by classifying each reference as a heap reference. The resulting analysis runs in polynomial time and has the same precision as PDCFA for $k = 0$. We conjecture that this variant is not a viable alternative in practice because of the significant loss in precision.

Kobayashi proposed a way to statically verify properties of functional programs using model-checking [36, 37]. He models a program by a higher-order recursion scheme $\mathcal{G}$, expresses the property of interest in the modal μ-calculus and checks if the infinite tree generated by $\mathcal{G}$ satisfies the property. This technique can be used for flow analysis, since flow analysis can be encoded as a model-checking problem, and it gives precise results. However, it only applies to pure, typed languages and it is computationally expensive.

9.4 Analyses employing big-step semantics

Wand and Steckler used flow analysis to prove the correctness of closure conversion [72]. The problem they tackle is that functional language compilers have an array of options for compiling lambdas: a lambda may never need to be closed over, or it may need a closure for some or all of its free variables. The authors use flow analysis to find all possible call sites for a procedure and check that each call site uses the calling protocol expected by the procedure. They use big-step semantics for their standard and non-standard concrete semantics. However, they perform flow analysis using constraint solving, not by abstract interpretation of the operational semantics.

The first analysis for functional languages based on abstract interpretation of big-step semantics was Serrano's formulation of 0CFA [58]. Serrano used 0CFA for optimizations in the Bigloo Scheme compiler. The analysis works on the direct-style AST and does not require labeling each subex-

pression in the program. Like Big CFA2, recursion in the analyzed program can cause non-termination in the analyzer. To avoid that, the analysis stops when it sees a duplicate function entry. Reppy's modular analysis for ML is based on Serrano's [51].

The most closely related work to Big CFA2 is an abstract-interpretation-based analysis for logic programs by Muthukumar and Hermenegildo [46]. Their analysis has similarities to Big CFA2, but in a more restricted setting, without higher-order functions and mutation. It uses call/success pairs, which are analogous to summaries. It is implemented recursively, so it uses the runtime stack to find return points quickly. For recursive predicates, the analysis avoids non-termination by doing a fixpoint computation.

Navas *et al.* [47, 41] showed how to translate Java bytecode to a Horn-clause-based intermediate representation, in order to apply Muthukumar and Hermenegildo's analysis to Java programs. The resulting analysis has several limitations compared to Big CFA2. Since the IR has no state, they model state using store-passing style; each Java statement s becomes a clause that takes as input all formal parameters and local variables of the method s appears in. Also, the translation uses type information, so it does not generalize gracefully to dynamic languages. First, virtual calls are translated away. Every virtual call is translated to several clauses, one for each method that can be called. This set of methods is approximated using class-hierarchy analysis. Second, the translation uses type information from method signatures to find which exceptions can be thrown, so it cannot handle runtime exceptions in Java (or exceptions in dynamic languages). Last, the analysis handles recursion in an ad-hoc way. It differentiates between recursion in a single method and mutually-recursive methods. For the latter, it requires recording dependencies between methods and tagging analysis results as "complete" or "approximate." In contrast, Big CFA2 treats all instances of recursion uniformly, including recursion that is not syntactically apparent in the source code.

9.5 JavaScript analyses

Guha *et al.* created a flow analysis for JavaScript for intrusion detection [25]. A writer of a client-side application can run the analysis to extract a model of expected behavior of the client, as seen from the server. The analysis uses

the model to create a proxy that detects malicious clients, *i.e.*, clients whose messages do not conform to the model. The authors use uniform kCFA [49] for the flow analysis part of their tool. Since the focus of this work is security, they do not describe the flow analysis aspect in detail. For example, they do not mention how they extend uniform kCFA to handle JavaScript objects, whether they model objects with context in the heap, the call-string length used in their experiments, *etc*. They do however say that the analysis tracks string literals and models string concatenation. The analysis can take several minutes for a few thousand lines of code. Memory requirements are not discussed.

Gatekeeper is a tool by Guarnieri and Livshits that uses flow analysis to check if JavaScript programs conform to certain security and reliability policies [24]. They express the flow-analysis relations in Datalog and use bddbddb [73] to compute the relations. Gatekeeper performs a whole-program analysis, so it disallows constructs that inject code at runtime such as `eval` and the `Function` constructor. It also disallows `with`. Guarnieri and Livshits define a large subset of JavaScript, called JavaScript$_{\mathrm{SAFE}}$, for which their analysis is sound. If the analysis finds a program to not belong in JavaScript$_{\mathrm{SAFE}}$, it attempts to insert runtime checks that enforce the policy. The analysis gives up on some programs because it cannot ensure that they follow the policy, even with runtime checks. Gatekeeper analyzes programs of similar size to the V8 benchmarks in a few seconds. Memory usage is not reported. Regarding the characteristics of the analysis, the authors do not model operations on scalars (not needed for the security policies of interest), and do not discuss deletion of properties. Also, the analysis does not keep track of the order of objects in the prototype chain. Objects are modeled by allocation site, without context. As far as we can tell, the analysis is monovariant.

Jensen *et al.* created a flow analysis for JavaScript type inference called TAJS [31, 32]. Their work draws inspiration from Thiemann and Heidegger's type systems [69, 28]. TAJS is sound and handles all of core JavaScript except `eval`. The authors go to great lengths to faithfully model JavaScript, and ensure soundness while maintaining precision. Unlike DoctorJS, TAJS uses sophisticated lattices for numbers and strings. It can also mark properties as "maybe there" to do property deletion soundly (see sec. 7.2). Recently, the authors extended the analysis to the DOM [30]. They model the

more commonly used parts of the DOM, but not all of it. For events, they do not track where listeners are attached in the DOM tree and do not model the capture and bubbling phases of event propagation. Since they do not know the order in which events will be fired, they analyze the listeners in a random order (but the load listeners before all others).

Algorithmically, TAJS is a descendant of the functional approach, like the iterative CFA2. CFA2 analyzes two calls to a function separately if the arguments differ, and creates two summaries. TAJS is more approximate, it separates the calls based only on the value of the receiver object. TAJS does not use a global heap with timestamps, every state has its own abstract heap. Objects are abstracted by allocation site without context, but TAJS employs a recency abstraction to increase precision [5]. To mitigate constantly copying heaps from one state to the other, the authors use a technique which they call "lazy propagation."

Jensen *et al.* implemented TAJS in Java and used it to find type-related bugs in JavaScript programs or prove the absence of them. Their results show that TAJS is precise but it does not scale. For example, it takes 136.7sec and 140.5MB for the V8 deltablue benchmark. For the SunSpider 3d-raytrace benchmark, it takes 8.2sec and 10.1MB (*cf.* table 8.2). With the DOM extensions, the largest program they analyzed is 2905LOC and TAJS takes 57.1sec. One justification for the performance of TAJS is that in the speed-precision-soundness trade-off the authors favored precision and soundness (*cf.* ch. 7). Another reason is the choice of not using a single global heap for all states. We believe that using the whole argument tuple for summaries has a bigger impact on precision than using one heap per state and is cheaper to implement.

CHAPTER 10

Future work

Polyvariance for heap-allocated data

Stack lookups make CFA2 polyvariant because different calls to the same function are analyzed in different environments. However, we allocate data in the heap without context, so CFA2 is monovariant in the heap.

CFA2 is strictly more precise than 0CFA because of stack lookups. For $k > 0$, CFA2 and kCFA are incomparably precise. In section 3.3.3, we saw an example that confuses kCFA for any k by repeated η-expansion; CFA2 is precise because there are no heap references. On the other hand, in the program

```
(let* ((f (λ (x) (λ () x)))
       (n1 ((f 1)))
       (n2 ((f 2))))
  (+ n1 n2)))
```

CFA2 finds that `n2` is bound to $\{1, 2\}$, but 1CFA finds the precise answer because `f` is called from two different call sites.

It is simple to extend CFA2 with call-strings polyvariance in the heap and get a family of analysis, CFA2.k. Then, any instance of CFA2.k would be strictly more precise than the corresponding instance of kCFA. However, we have argued that call strings are not a good choice of context, so we do not believe that this construction would have practical utility.

The use of stack references and the treatment of calls and returns in CFA2 are orthogonal to the choice of context in the heap. Currently, DoctorJS abstracts objects by allocation site, but we would like to experiment with more precise options from the literature.

Better summaries

CFA2 creates summaries in a conservative manner. Suppose that during the analysis we find a call to a function f with argument a and we already have a summary for f with a. If the heap has changed since the creation of the summary, we reanalyze. However, the changes to the heap may be in parts that f never touches. In this case, it is safe to reuse the summary.

One way to increase summary reuse is to identify functions that never touch the heap at all and reuse summaries for these functions. Note that a pure function may still touch the heap if its body contains a heap reference. Also, a non-pure function may not touch the heap if the side effects are to stack variables only.

A more fine-grained solution, but potentially harder to implement, is to track which parts of the heap are relevant for each function and allow summary reuse when the changes in the heap do not affect the relevant parts. The symbolic analysis of Chandra *et al.* uses this approach [8]. Lazy propagation in TAJS is also in this spirit [32].

It is plausible that we can use these ideas to increase summary reuse and speed up DoctorJS. However, it is also likely that the overhead from tracking the relevant parts of the heap for each function is bigger than the gain from the summary reuse. Avoiding reanalysis is more crucial in TAJS than in CFA2 because TAJS uses one heap per state.

First-class control

The algorithm for CFA2 with first-class control is not complete, so we lose precision when going from abstract to local semantics (sec. 5.3.3). An interesting question is whether incompleteness can be avoided. We would like to find a complete algorithm or prove that the abstract semantics corresponds to a machine whose reachability is not decidable.

Big CFA2 does not handle first-class control, so DoctorJS currently models generators in an unsound way. One way to model generators correctly is to adapt the algorithm of Fig. 5.2 for Big CFA2. This way, generators would induce spurious control flows because the algorithm merges possibly escaping continuations. But generators, like exceptions, are strictly less powerful than first-class continuations. Thus, there may be a specialized way to model

them correctly in Big CFA2 without loss in precision.

DoctorJS implementation improvements

We have mentioned that DoctorJS abstracts objects by allocation site. This abstraction is sometimes imprecise, *e.g.*, when objects are created using factory methods. The raytrace example in section 8.2.1 shows that heap specialization may improve speed alongside precision. For these reasons, we want to investigate more precise alternatives for abstracting objects.

Representing the abstract heap as a map from addresses to values causes the analysis to miss relations between heap-allocated data and do redundant work (sec. 8.2.2). We want to try a graph-based representation of points-to relations. With a graph, when a flow set grows, we can propagate the change to the relevant parts of the heap immediately and do fewer timestamp increases. Naturally, a graph representation raises other issues, such as cycle elimination [18] and whether or not to add transitive edges [9, 29].

DoctorJS uses one frame to analyze the body of a function flow insensitively. This is fast and precision is good most of the time. However, in some cases it is important to analyze the branches of an `if` statement in different environments, *e.g.*, when the test expression is checking for `null`. We can improve precision in these cases by using SSA [27].

Analysis clients

This dissertation shows how to create precise pushdown analysis techniques for expressive, higher-order languages. We discussed some applications in chapter 8, but there many more interesting possibilities.

CFA2 is well suited for escape analysis to convert heap allocations to stack allocations. Besides the optimization benefit, the information from an escape analysis benefits the flow analysis itself because more data stay in frames and do not leak in the (less precise) abstract heap.

CFA2 could also improve environment analyses [61, 42]. The use of stack references is a simple syntactic way to prove that two references are bound in the same environment. An important application of environment analysis is optimizing the representation of closures [38, 72, 42].

CFA2 could also help with static debugging, especially in dynamic lan-

guages. Current IDEs for dynamic languages offer little debugging support to programmers. A flow analysis can find many type-related bugs: calling a function with the wrong number or type of arguments, accessing non-existent properties of objects, dereferencing `null`, *etc*. Type information provided by DoctorJS can catch many such errors early.

CHAPTER 11

Conclusions

In this dissertation, we have proposed techniques for higher-order flow analysis with unbounded call/return matching. By modeling calls and returns faithfully, it becomes possible to overcome the limitations of finite-state analyses. In addition, our analyses are more broadly applicable than previous pushdown analyses for higher-order languages.

The proposed techniques do not depend on a particular dataflow lattice. As a result, they can be used to improve the precision of any existing dataflow analysis for higher-order languages. In addition, they are well suited for environment analysis and stack-based optimizations.

Several independent parameters in CFA2 could be tuned to change precision and speed: stack filtering, reuse/widening of summaries, flow/path sensitivity. We would like to perform experiments with various combinations of these parameters, in order to quantify the effect of each one on precision and speed, and find combinations that strike a good balance in practice.

Besides the modeling of control flow, the choice of abstraction for heap-allocated data (closure environments, objects, records, *etc*.) plays a major role in a flow analysis. The present work has not addressed this issue. The proposed techniques are orthogonal to the choice of heap abstraction and can be combined with existing solutions.

We believe that pushdown models are better suited for higher-order flow analysis than finite-state models. Our experiments show that call/return matching can yield precise and fast analyses in practice. Additional experimentation is needed to quantify the benefits of pushdown analyses better. It would be instructive to measure the precision and speed of CFA2 on several client optimizations and compare it to kCFA and other finite-state analyses.

This way, we can see the precision differences across various clients and understand which clients benefit the most from a pushdown abstraction.

Our vision is that precise, polyvariant flow analysis should be efficient. We believe it is possible to consistently analyze programs up to a hundred thousand lines of code in a few seconds. CFA2 is a step in this direction.

APPENDIX A

Proofs

Lemma. Let ς be a reachable concrete state of the form $(\ldots, ve, t)$. Then,

1. For any closure $(lam, \rho) \in \text{range}(ve)$, it holds that $\text{dom}(\rho) \cap BV(lam) = \emptyset$.
2. If ς is an $Eval$ state $(call, \rho, ve, t)$, then $\text{dom}(\rho) \cap BV(call) = \emptyset$.
3. If ς is an $Apply$, any closure (lam, ρ) in operator or argument position satisfies $\text{dom}(\rho) \cap BV(lam) = \emptyset$.

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.

Show for $\mathcal{I}(pr)$***:***
$\mathcal{I}(pr)$ is a $UApply$ of the form $((pr, \emptyset), (lam, \emptyset), halt, \emptyset, \langle\rangle)$. (1) holds because ve is $\emptyset$. (3) holds because both closures have empty environments.

Rule [UEA]***:***
The [UEA] transition is:
$([\![(f\ e\ q)^l]\!], \rho, ve, t) \to (\mathcal{A}(f, \rho, ve), \mathcal{A}(e, \rho, ve), \mathcal{A}(q, \rho, ve), ve, l :: t)$

(1) holds for ς' because ve does not change in the transition. The operator of ς' is a closure (lam, ρ'). We must show that $\text{dom}(\rho') \cap BV(lam) = \emptyset$.
If $Lam_?(f)$, then $lam = f$ and $\rho' = \rho$. Also, we know
$\text{dom}(\rho) \cap BV([\![(f\ e\ q)^l]\!]) = \emptyset$
$\Rightarrow\ \text{dom}(\rho) \cap (BV(f) \cup BV(e) \cup BV(q)) = \emptyset$
$\Rightarrow\ \text{dom}(\rho) \cap BV(f) = \emptyset$.
If $Var_?(f)$, then $(lam, \rho') \in \text{range}(ve)$, so we get the desired result because ve satisfies (1).
Similarly for $\pi_2(\hat{\varsigma}')$ and $\pi_3(\hat{\varsigma}')$.

Rule [UAE]***:***
The [UAE] transition is:
$(\langle [\![(\lambda_l (u\ k)\ \ call)]\!], \rho\rangle, d, c, ve, t) \to (call, \rho', ve', t)$
$\rho' = \rho[u \mapsto t,\ k \mapsto t\,]$
$ve' = ve[(u,t) \mapsto d,\ (k,t) \mapsto c\,]$

To show (1) for ve', we must show that d and c do not violate the property. Let d be (lam_1, ρ_1). Since ς satisfies (3), we know $\mathrm{dom}(\rho_1) \cap BV(lam_1) = \emptyset$, 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 variable names are unique. Also, from property (3) for ς we know $\mathrm{dom}(\rho) \cap BV([\![(\lambda_l (u\ k)\ \ call)]\!]) = \emptyset$, which implies $\mathrm{dom}(\rho) \cap BV(call) = \emptyset$. We must show $\mathrm{dom}(\rho') \cap BV(call) = \emptyset$
$\Leftrightarrow\ (\mathrm{dom}(\rho) \cup \{u,k\}) \cap BV(call) = \emptyset$
$\Leftrightarrow\ (\mathrm{dom}(\rho) \cap BV(call)) \cup (\{u,k\} \cap BV(call)) = \emptyset$
$\Leftrightarrow\ \emptyset \cup \emptyset = \emptyset.$

Other rules:
Similarly for the other two transitions. □

A.1 Proofs for CFA2 without first-class control

In this section, we prove theorems 9, 13 and 14 for the stack-filtering semantics. Without stack filtering, the proofs are similar and simpler.

To make it easier for the reader to navigate the structure of the proofs and understand what facts each theorem or lemma relies on, we show a (directed

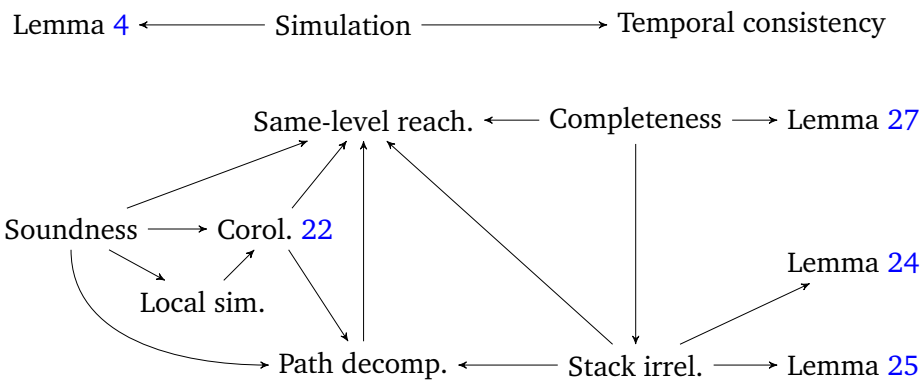


Figure A.1: Proof dependencies

acyclic) graph of the dependencies between the proofs in fig. A.1. The graph has edges only for the immediate dependencies, *i.e.*, when a theorem or lemma mentions another one in the proof.

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

Proof. By cases on the concrete transition.

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

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

The abstract transition is
$([\![(f\ e\ q)^l]\!], st, h) \rightsquigarrow (f', \hat{d}, \hat{c}, st', h)$
$f' \in \hat{\mathcal{A}}_{\text{u}}(f, l, st, h)$
$\hat{d} = \hat{\mathcal{A}}_{\text{u}}(e, l, st, h)$
$\hat{c} = \hat{\mathcal{A}}_{\text{k}}(q, st)$

$$st' = \begin{cases} pop(st) & Var_?(q) \\ st & Lam_?(q) \wedge (H_?(l, f) \vee Lam_?(f)) \\ st[f \mapsto \{f'\}\,] & Lam_?(q) \wedge S_?(l, f) \end{cases}$$

$\hat{\varsigma}$ has many possible successors, one for each lambda in $\hat{\mathcal{A}}_{\text{u}}(f, l, st, h)$. We must show that one of them is a state $\hat{\varsigma}'$ such that $|\varsigma'|_{\text{ca}} \sqsubseteq \hat{\varsigma}'$.

The variable environment and the heap do not change in the transitions, so for ς' and $\hat{\varsigma}'$ we know that $|ve|_{\text{ca}} \sqsubseteq h$. We must show $\pi_1(proc) = f'$, $|d|_{\text{ca}} \sqsubseteq \hat{d}$, $|c|_{\text{ca}} \sqsubseteq \hat{c}$ and $ts' \sqsubseteq st'$, where ts' is the stack of $|\varsigma'|_{\text{ca}}$.
We first show $\pi_1(proc) = f'$, by cases on f:

- $Lam_?(f)$
  Then, $proc = (f, \rho)$ and $f' \in \{f\}$, so $f' = f$.
- $S_?(l, f)$
  Then, $proc = ve(f, \rho(f))$, a closure of the form (lam, ρ').

Since $ts(f) = |ve(f, \rho(f))|_{\text{ca}} = \{lam\}$ and $ts \sqsubseteq st$, we get $lam \in st(f)$. So, we pick f' to be lam.

- $H_?(l, f)$
  Then, $proc = ve(f, \rho(f))$, a closure of the form (lam, ρ').
  Since $|ve|_{\text{ca}} \sqsubseteq h$ and $lam \in |ve|_{\text{ca}}(f)$, we get $lam \in h(f)$.
  So, we pick f' to be lam.

Showing $|d|_{\text{ca}} \sqsubseteq \hat{d}$ is similar. We now show $|c|_{\text{ca}} \sqsubseteq \hat{c}$, by cases on q:

- $Lam_?(q)$
  Then, $c = (q, \rho)$ and $\hat{c} = q$, so $|c|_{\text{ca}} \sqsubseteq \hat{c}$.
- $Var_?(q)$ and $c = ve(q, \rho(q)) = halt$
  Then, $ts(q) = halt$. Since $ts \sqsubseteq st$, we get $st(q) = halt$. Thus, $\hat{c} = halt$.
- $Var_?(q)$ and $c = ve(q, \rho(q)) = (lam, \rho')$
  Similar to the previous case.

It remains to show that $ts' \sqsubseteq st'$. We proceed by cases on q and f:

- $Var_?(q)$ and $c = ve(q, \rho(q)) = halt$
  Then, $ts' = \langle\rangle$. By $ts \sqsubseteq st$, we know that ts and st have the same size. Also, $st' = pop(st)$, thus $st' = \langle\rangle$. Therefore, $ts' \sqsubseteq st'$.
- $Var_?(q)$ and $c = ve(q, \rho(q)) = (lam, \rho')$
  By fig. 4.3, we know that $ts' = toStack(LV(\mathcal{L}(lam)), \rho', ve) = pop(ts)$. Also, $st' = pop(st)$. Thus, to show $ts' \sqsubseteq st'$ it suffices to show $pop(ts) \sqsubseteq pop(st)$, which holds because $ts \sqsubseteq st$.
- $Lam_?(q) \wedge (Lam_?(f) \vee H_?(l, f))$
  Then, $ts' = ts$ and $st' = st$, so $ts' \sqsubseteq st'$.
- $Lam_?(q) \wedge S_?(l, f)$
  By $LV(\mathcal{L}(q)) = LV(l)$, we get that $ts' = ts$. Also, $proc = ve(f, \rho(f))$, a closure of the form (lam, ρ'). We pick f' to be lam. The stack of $\hat{\varsigma}'$ is $st' = st[f \mapsto \{lam\}]$. Since $pop(ts) \sqsubseteq pop(st)$, we only need to show that the top frames of ts' and st' are in $\sqsubseteq$. For this, it suffices to show that $ts'(f) \sqsubseteq st'(f)$ which holds because $ts'(f) = ts(f) = \{lam\}$.

Rule [UAE]***:***

$(\langle [\![(\lambda_l (u\ k)\ call)]\!], \rho\rangle, d, c, ve, t) \to (call, \rho', ve', t)$

$\rho' = \rho[u \mapsto t,\ k \mapsto t\,]$

$ve' = ve[(u, t) \mapsto d,\ (k, t) \mapsto c\,]$

Let $ts = \begin{cases} \langle\rangle & c = halt \\ toStack(LV(\mathcal{L}(lam)), \rho_1, ve) & c = (lam, \rho_1) \end{cases}$
Since $|\varsigma|_{\text{ca}} \sqsubseteq \hat{\varsigma}$, $\hat{\varsigma}$ is of the form $(\llbracket (\lambda_l (u\ k)\ call) \rrbracket, \hat{d}, \hat{c}, st, h)$,
where $|d|_{\text{ca}} \sqsubseteq \hat{d}$, $|c|_{\text{ca}} = \hat{c}$, $ts \sqsubseteq st$ and $|ve|_{\text{ca}} \sqsubseteq h$.

The abstract transition is
$(\llbracket (\lambda_l (u\ k)\ call) \rrbracket, \hat{d}, \hat{c}, st, h) \rightsquigarrow (call, st', h')$
$st' = push([u \mapsto \hat{d},\ k \mapsto \hat{c}\,], st)$
$h' = \begin{cases} h \sqcup [u \mapsto \hat{d}\,] & H_?(u) \\ h & S_?(u) \end{cases}$

Let ts' be the stack of $|\varsigma'|_{\text{ca}}$.
From $iu_\lambda(call) = \lambda_l$, we get $ts' = toStack(LV(l), \rho', ve')$.
We must show that $|\varsigma'|_{\text{ca}} \sqsubseteq \hat{\varsigma}'$, *i.e.*, $ts' \sqsubseteq st'$ and $|ve'|_{\text{ca}} \sqsubseteq h'$.

We assume that $c = (lam, \rho_1)$ and that $H_?(u)$ holds, the other cases are simpler. In this case, $|ve'|_{\text{ca}}$ is the same as $|ve|_{\text{ca}}$ except that $|ve'|_{\text{ca}}(u) = |ve|_{\text{ca}}(u) \sqcup |d|_{\text{ca}}$. Also, $h'(u) = h(u) \sqcup \hat{d}$, thus $|ve'|_{\text{ca}} \sqsubseteq h'$.

We know that ρ' contains bindings for u and k, and by lemma 4 it does not bind any variables in $BV(call)$. Since $LV(l) \setminus \{u, k\} \subseteq BV(call)$, ρ' does not bind any variables in $LV(l) \setminus \{u, k\}$.
Thus, the top frame of ts' is $[u \mapsto |d|_{\text{ca}},\ k \mapsto |c|_{\text{ca}}\,]$. 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 $ts' \sqsubseteq st'$, we must show that $pop(ts') \sqsubseteq pop(st') \Leftrightarrow\ pop(ts') \sqsubseteq st \Leftarrow\ pop(ts') = ts$.
We know that $pop(ts') = toStack(LV(\mathcal{L}(lam)), \rho_1, ve')$ and
$ts = toStack(LV(\mathcal{L}(lam)), \rho_1, ve)$. By lemma 6, $pop(ts')$ will not contain the two bindings born at time t because they are younger than all bindings in ρ_1. This implies that $pop(ts') = ts$.

Rule [CEA]***:***
$(\llbracket (q\ e)^\gamma \rrbracket, \rho, ve, t) \rightarrow (proc, d, ve, \gamma :: t)$
$proc = \mathcal{A}(q, \rho, ve)$
$d = \mathcal{A}(e, \rho, ve)$

Let $ts = toStack(LV(\gamma), \rho, ve)$. Since $|\varsigma|_{\text{ca}} \sqsubseteq \hat{\varsigma}$, $\hat{\varsigma}$ is of the form $(\llbracket (q\ e)^\gamma \rrbracket, st, h)$,
where $|ve|_{\text{ca}} \sqsubseteq h$ and $ts \sqsubseteq st$.
The abstract transition is

$([\![(q\ e)^{\gamma}]\!], st, h) \rightsquigarrow (q', \hat{d}, st', h)$
$q' = \hat{\mathcal{A}}_{\mathrm{k}}(q, st)$
$\hat{d} = \hat{\mathcal{A}}_{\mathrm{u}}(e, \gamma, st, h)$
$st' = \begin{cases} pop(st) & Var_?(q) \\ st & Lam_?(q) \end{cases}$

Let ts' be the stack of $|\varsigma'|_{\mathrm{ca}}$. We must show that $|\varsigma'|_{\mathrm{ca}} \sqsubseteq \hat{\varsigma}'$, *i.e.*, $|proc|_{\mathrm{ca}} = q'$, $|d|_{\mathrm{ca}} \sqsubseteq \hat{d}$, and $ts' \sqsubseteq st'$.
We first show $|proc|_{\mathrm{ca}} = q'$, by cases on q:

- $Lam_?(q)$
  Then, $proc = (q, \rho)$ and $q' = q$. Thus, $|proc|_{\mathrm{ca}} = q'$.
- $Var_?(q)$ and $proc = ve(q, \rho(q)) = (lam, \rho_1)$
  Since $q \in LV(\gamma)$ we get $ts(q) = lam$. From this and $ts \sqsubseteq st$, we get $st(q) = lam$, which implies $q' = lam$, which implies $|proc|_{\mathrm{ca}} = q'$.
- $Var_?(q)$ and $proc = ve(q, \rho(q)) = halt$
  Similar to the previous case.

Showing $|d|_{\mathrm{ca}} \sqsubseteq \hat{d}$ is similar, by cases on e.
Last, we show $ts' \sqsubseteq st'$, by cases on q:

- $Lam_?(q)$
  Then, $st' = st$. Also, $ts' = toStack(LV(\mathcal{L}(q)), \rho, ve)$ and $LV(\mathcal{L}(q)) = LV(\gamma)$. Thus, $ts' = ts$, which implies $ts' \sqsubseteq st'$.
- $Var_?(q)$ and $proc = ve(q, \rho(q)) = (lam, \rho_1)$
  Then, $ts' = toStack(LV(\mathcal{L}(lam)), \rho_1, ve) = pop(ts)$ and $st' = pop(st)$. To show $ts' \sqsubseteq st'$, it suffices to show $pop(ts) \sqsubseteq pop(st)$, which holds by $ts \sqsubseteq st$.
- $Var_?(q)$ and $proc = ve(q, \rho(q)) = halt$
  Similar to the previous case.

Rule [CAE]***:***
This case requires arguments similar to the previous cases. □

Definition 19 (Push Monotonicity)**.**
Let $p = \hat{\varsigma}_e \rightsquigarrow^* \hat{\varsigma}$ where $\hat{\varsigma}_e$ is an entry with stack st_e. The path p is **push monotonic** iff every transition $\hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}_2$ satisfies the following property:

> If the stack of $\hat{\varsigma}_1$ is st_e then the transition can only push the stack, it cannot pop or modify the top frame.

Push monotonicity is a property of paths, not of individual transitions. A push monotonic path can contain transitions that pop, as long as the stack never shrinks below the stack of the initial state of the path. The following properties are simple consequences of push monotonicity.

- Every path of the form $\hat{\mathcal{I}}(pr) \leadsto^* \hat{\varsigma}$ is push monotonic.
- Every prefix of a push-monotonic path is push monotonic.
- The stack of the first state in a push-monotonic path is a suffix of the stack of every other state in the path.

Lemma 20 (Same-level reachability)**.** Let $\hat{\varsigma}_e = (\llbracket (\lambda_l (u\ k)\ call) \rrbracket, \hat{d}, \hat{c}, st_e, h_e)$, $\hat{\varsigma} = (\ldots, st, h)$ and $p = \hat{\varsigma}_e \leadsto^* \hat{\varsigma}$, where $\hat{\varsigma}_e \in CE^*_p(\hat{\varsigma})$.

1. If $\hat{\varsigma}$ is an entry then $st = st_e$ and the continuation argument of $\hat{\varsigma}$ is $\hat{c}$.
2. If $\hat{\varsigma}_e \in (CE^*_p(\hat{\varsigma}) \setminus \{CE_p(\hat{\varsigma})\})$ then

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

   If $\hat{\varsigma}_e = CE_p(\hat{\varsigma})$ then

   - there is a frame tf such that $st = tf :: st_e$, $\mathrm{dom}(tf) \subseteq LV(l)$, $tf(u) \sqsubseteq \hat{d}$, $tf(k) = \hat{c}$.
   - 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 $\gamma \in LL(l)$.

3. p is push monotonic.

Note This lemma shows some intuitive properties about a state $\hat{\varsigma}$ and its corresponding entry $\hat{\varsigma}_e$. First, since $\hat{\varsigma}$ is in the procedure whose entry is $\hat{\varsigma}_e$, the top frame in $\hat{\varsigma}$ binds variables in this procedure, *i.e.*, $\mathrm{dom}(tf) \subseteq LV(l)$. Also, the return continuation of $\hat{\varsigma}$ is the continuation passed to $\hat{\varsigma}_e$, *i.e.*, $tf(k) = \hat{c}$. Last, the path from $\hat{\varsigma}_e$ to $\hat{\varsigma}$ is obviously push monotonic, because we must return in order to get a stack smaller than st_e. And if $\hat{\varsigma}_e$ is not $CE_p(\hat{\varsigma})$, but is in $CE^*_p(\hat{\varsigma})$, similar things hold.

Proof. By induction on the length of p. Note that (3) follows from the form of the stack in (1) and (2), so we will not prove it separately.
If the length of p is 0 then $\hat{\varsigma} = \hat{\varsigma}_e$ so the lemma trivially holds.
If the length is greater than 0, we take two cases.

Case 1: $\hat{\varsigma}_e = CE_p(\hat{\varsigma})$
In this case, since $\hat{\varsigma}$ is not an entry, the second or third branch of def. 11 determine the shape of p.

- $p = \hat{\varsigma}_e \rightsquigarrow^* \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$
  Here, the predecessor $\hat{\varsigma}'$ of $\hat{\varsigma}$ is not an Exit-Ret, and $\hat{\varsigma}_e = CE_p(\hat{\varsigma}')$. We proceed by cases on $\hat{\varsigma}'$. Note that $\hat{\varsigma}'$ cannot be a $\widehat{UEval}$ because then $\hat{\varsigma}$ is an entry, so $\hat{\varsigma} = CE_p(\hat{\varsigma})$, and our assumption that $\hat{\varsigma}_e = CE_p(\hat{\varsigma})$ breaks.
  - $\hat{\varsigma}'$ is an inner $\widehat{CEval}$ of the form $([\![(clam\ e)^\gamma]\!], st', h')$.
    By *IH*, $st' = tf' :: st_e$, $\mathrm{dom}(tf') \subseteq LV(l)$, $tf'(u) \sqsubseteq \hat{d}$, $tf'(k) = \hat{c}$ and $\gamma \in LL(l)$. By rule $\widehat{[\mathrm{CEA}]}$, $\hat{\varsigma} = (clam, \hat{d}', st', h')$. From $\gamma \in LL(l)$ we get $\mathcal{L}(clam) \in LL(l)$. Also, the stack is unchanged in the transition so st satisfies the required properties.
  - $\hat{\varsigma}'$ is a $\widehat{CApply}$ of the form $([\![(\lambda_\gamma(u')\ call')]\!], \hat{d}', st', h')$.
    By *IH*, $st' = tf' :: st_e$, $\mathrm{dom}(tf') \subseteq LV(l)$, $tf'(u) \sqsubseteq \hat{d}$, $tf'(k) = \hat{c}$, $\gamma \in LL(l)$.
    By rule $\widehat{[\mathrm{CAE}]}$, $\hat{\varsigma} = (call', tf :: st_e, h)$ where $tf = tf'[u' \mapsto \hat{d}']$.
    From $\gamma \in LL(l)$ we get $u' \in LV(l)$, so $\mathrm{dom}(tf) = \mathrm{dom}(tf') \cup \{u'\} \subseteq LV(l)$.
    Since variable names are unique, $u \neq u'$, so $tf(u) = tf'(u) \sqsubseteq \hat{d}$.
    Also, $tf(k) = tf'(k) = \hat{c}$. Last, from $\gamma \in LL(l)$ we get $\mathcal{L}(call') \in LL(l)$.
  - $\hat{\varsigma}'$ is a $\widehat{UApply}$
    Then, $\hat{\varsigma}' = \hat{\varsigma}_e$ because $\hat{\varsigma}_e = CE_p(\hat{\varsigma}')$. This case is simple.

- $p = \hat{\varsigma}_e \rightsquigarrow^+ \hat{\varsigma}_2 \rightsquigarrow \hat{\varsigma}_3 \rightsquigarrow^+ \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$
  Here, the third branch of def. 11 determines the shape of p, so $\hat{\varsigma}_2$ is a call, $\hat{\varsigma}_e = CE_p(\hat{\varsigma}_2)$, $\hat{\varsigma}'$ is an Exit-Ret and $\hat{\varsigma}_3 \in CE_p^*(\hat{\varsigma}')$.
  By *IH* for $\hat{\varsigma}_e \rightsquigarrow^+ \hat{\varsigma}_2$, we get $\hat{\varsigma}_2 = ([\![(f\ e\ clam)^{l_2}]\!], tf_2 :: st_e, h_2)$, where $\mathrm{dom}(tf_2) \subseteq LV(l)$, $tf_2(u) \sqsubseteq \hat{d}$, $tf_2(k) = \hat{c}$ and $l_2 \in LL(l)$.
  By rule $\widehat{[\mathrm{UEA}]}$, we get $\hat{\varsigma}_3 = (ulam, \hat{d}_3, \hat{c}_3, tf_3 :: st_e, h_2)$ where
  $$tf_3 = \begin{cases} tf_2[f \mapsto \{ulam\}] & S_?(l_2, f) \\ tf_2 & H_?(l_2, f) \vee Lam_?(f) \end{cases}$$
  We only show the lemma when $S_?(l_2, f)$, the other case is simpler.
  By *IH* for $\hat{\varsigma}_3 \rightsquigarrow^+ \hat{\varsigma}'$, $\hat{\varsigma}' = (call', tf' :: (tf_3 :: st_e), h')$, where $tf'(k') = clam$.
  By rule $\widehat{[\mathrm{CEA}]}$, we get $\hat{\varsigma} = (clam, \hat{d}', tf_3 :: st_e, h')$.
  From $l_2 \in LL(l)$ we get $\mathcal{L}(clam) \in LL(l)$.

Also, $S_?(l_2, f)$ implies $f \in LV(l)$, so $\mathrm{dom}(tf_3) = \mathrm{dom}(tf_2) \cup \{f\} \subseteq LV(l)$. Last, we take cases depending on whether u and f are the same variable. If $u = f$, $tf_3(u) = \{ulam\} \sqsubseteq tf_2(u) \sqsubseteq \hat{d}$. If $u \neq f$, $tf_3(u) = tf_2(u) \sqsubseteq \hat{d}$.

Case 2: $\hat{\varsigma}_e \neq CE_p(\hat{\varsigma})$
Then, by the second branch of def. 12, p has the form $\hat{\varsigma}_e \rightsquigarrow^+ \hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^* \hat{\varsigma}$, where $\hat{\varsigma}_1$ is a tail call, $\hat{\varsigma}_2 = CE_p(\hat{\varsigma})$ and $\hat{\varsigma}_e \in CE_p^*(\hat{\varsigma}_1)$.
By *IH* for $\hat{\varsigma}_e \rightsquigarrow^+ \hat{\varsigma}_1$, the stack of $\hat{\varsigma}_1$ has the form $tf_1 :: st_e$ where $tf_1(k') = \hat{c}$.
By rule $[\widehat{\mathrm{UEA}}]$, $\hat{\varsigma}_2$ has the form $(\ldots, \ldots, \hat{c}, st_e, \ldots)$.

- If $\hat{\varsigma}$ is an entry, then $\hat{\varsigma} = \hat{\varsigma}_2$, so $\hat{\varsigma}$ has the required form (item 1).
- If $\hat{\varsigma}$ is not an entry, by *IH* for $\hat{\varsigma}_2 \rightsquigarrow^+ \hat{\varsigma}$, there are tf and k'' such that $st = tf :: st_e$ and $tf(k'') = \hat{c}$. Thus, $\hat{\varsigma}$ has the required form. □

The following lemma states that, in push monotonic paths, the corresponding entries of states are defined (with one exception). Every path from $\hat{\mathcal{I}}(pr)$ is push monotonic. Thus, when a program pr executes under the abstract semantics, each non-final state has a corresponding entry.

Lemma 21 (Path decomposition)**.**
Let $\hat{\varsigma}_e = (ulam, \hat{d}, \hat{c}, st_e, h_e)$ and let path $p = \hat{\varsigma}_e \rightsquigarrow^* \hat{\varsigma}$ be push-monotonic.

- If $\hat{\varsigma}$ is a $\widehat{CApply}$ of the form $(\hat{c}, \ldots, st_e, \ldots)$ then $CE_p(\hat{\varsigma})$ is not defined.
- Otherwise, p has the form $\hat{\varsigma}_e \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow^* \hat{\varsigma}$, where $\hat{\varsigma}_1 = CE_p(\hat{\varsigma})$.

Proof. We prove the lemma by induction on the length of p.
The base case is simple. If the length of p is greater than 0, then p has the form $\hat{\varsigma}_e \rightsquigarrow^* \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$. We takes cases on $\hat{\varsigma}'$.

$\hat{\varsigma}'$ is an Exit-Ret:
By *IH*, $CE_p(\hat{\varsigma}')$ is defined, so $CE_p^*(\hat{\varsigma}') \neq \emptyset$.

- If $\hat{\varsigma}_e \in CE_p^*(\hat{\varsigma}')$
  By lemma 20 for $\hat{\varsigma}_e \rightsquigarrow^+ \hat{\varsigma}'$ and rule $[\widehat{\mathrm{CEA}}]$, $\hat{\varsigma}$ has the form $(\hat{c}, \ldots, st_e, \ldots)$. We must show that $CE_p(\hat{\varsigma})$ is not defined. Assume that $CE_p(\hat{\varsigma})$ is defined. This can only happen if the third item of def. 11 applies. Then, p has the form $\hat{\varsigma}_e \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}_2 \rightsquigarrow \hat{\varsigma}_3 \rightsquigarrow^+ \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$, where $\hat{\varsigma}_2$ is a call, $CE_p(\hat{\varsigma}_2) = \hat{\varsigma}_1$ and $\hat{\varsigma}_3 \in CE_p^*(\hat{\varsigma}')$. By lemma 20 for $\hat{\varsigma}_3 \rightsquigarrow^+ \hat{\varsigma}'$, the stack of $\hat{\varsigma}_3$ is st_e, and by rule $[\widehat{\mathrm{UEA}}]$, the stack of $\hat{\varsigma}_2$ has the same size as st_e. But then by lemma 20

for $\hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}_2$, the stack of $\hat{\varsigma}_1$ is smaller than st_e, which is a contradiction because p is push monotonic. Therefore, $CE_p(\hat{\varsigma})$ is not defined.

- If $\hat{\varsigma}_e \notin CE_p^*(\hat{\varsigma}')$
  Let $\hat{\varsigma}_2$ be the earliest state in $CE_p^*(\hat{\varsigma}')$. Its predecessor $\hat{\varsigma}_1$ is a $\widehat{UEval}$. The path becomes $\hat{\varsigma}_e \rightsquigarrow^+ \hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^+ \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$. By *IH*, $CE_p(\hat{\varsigma}_1)$ is defined. Thus, $\hat{\varsigma}_1$ is not a tail call, because if it was then $CE_p(\hat{\varsigma}_1)$ would be in $CE_p^*(\hat{\varsigma}')$ and earlier than $\hat{\varsigma}_2$. By the third item of def. 11, $CE_p(\hat{\varsigma}) = CE_p(\hat{\varsigma}_1)$.

$\hat{\varsigma}'$ is a $\widehat{UEval}$:
Then, $CE_p(\hat{\varsigma}) = \hat{\varsigma}$.

$\hat{\varsigma}'$ is none of the above:
Then, $\hat{\varsigma}'$ is an entry, an inner $\widehat{CEval}$ or a $\widehat{CApply}$. By *IH*, $CE_p(\hat{\varsigma}')$ is defined. By the second item of def. 11, $CE_p(\hat{\varsigma}) = CE_p(\hat{\varsigma}')$. □

Corollary 22. Every path from $\hat{\mathcal{I}}(pr)$ is push monotonic. Thus, lemmas 20 and 21 imply that the stack of an $\widehat{Eval}$ state in $\mathcal{RS}$ is not empty.

Lemma 23 (Local simulation)**.**
Let $\hat{\varsigma} \in \mathcal{RS}$ and $\hat{\varsigma} \rightsquigarrow \hat{\varsigma}'$. If $\hat{\varsigma}$ is not an Exit-Ret then $|\hat{\varsigma}'|_{\text{al}} \in succ(|\hat{\varsigma}|_{\text{al}})$.

Proof. By cases on the abstract transition.
We only show the lemma for $[\widehat{\text{UEA}}]$, the other cases are similar.
$([\![(f\ e\ q)^l]\!], st, h) \rightsquigarrow (ulam, \hat{d}, \hat{c}, st', h)$
where $ulam \in \hat{\mathcal{A}}_{\text{u}}(f, l, st, h)$, $\hat{d} = \hat{\mathcal{A}}_{\text{u}}(e, l, st, h)$, $\hat{c} = \hat{\mathcal{A}}_{\text{k}}(q, st)$ and

$$st' = \begin{cases} pop(st) & Var_?(q) \\ st & Lam_?(q) \wedge (H_?(l, f) \vee Lam_?(f)) \\ st[f \mapsto \{ulam\}] & Lam_?(q) \wedge S_?(l, f) \end{cases}$$

By corollary 22, $st = tf :: st''$, so $|st|_{\text{al}} = tf \restriction UVar$.
Also, $|\hat{\varsigma}|_{\text{al}} = ([\![(f\ e\ q)^l]\!], |st|_{\text{al}}, h)$ and $|\hat{\varsigma}'|_{\text{al}} = (ulam, \hat{d}, h)$.
We must show that $ulam \in \tilde{\mathcal{A}}_{\text{u}}(f, l, |st|_{\text{al}}, h)$.
If $S_?(l, f)$ then $ulam \in tf(f)$, so $ulam \in (tf \restriction UVar)(f)$ because $f \in UVar$.
If $H_?(l, f)$ or $Lam_?(f)$, similarly.
Showing that $\hat{d} = \tilde{\mathcal{A}}_{\text{u}}(e, l, |st|_{\text{al}}, h)$ is similar. □

Theorem (Soundness)**.** Let $p = \hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}$. Then, after summarization:

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

Proof. By induction on the length of p.
If the length is 0, then $\hat{\mathcal{I}}(pr) \rightsquigarrow^0 \hat{\mathcal{I}}(pr)$. We know that $(\tilde{\mathcal{I}}(pr), \tilde{\mathcal{I}}(pr)) \in \mathit{Seen}$.
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:
Then, $CE_p(\hat{\varsigma}) = \hat{\varsigma}$. Also, $\hat{\varsigma}'$ is a call or a tail call.
By lemma 21, $p = \hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$, where $\hat{\varsigma}_1 = CE_p(\hat{\varsigma}')$.
By *IH*, $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}}) \in \mathit{Seen}$ which means that it has been entered in W and examined. By lemma 23, $|\hat{\varsigma}|_{\text{al}} \in \mathit{succ}(|\hat{\varsigma}'|_{\text{al}})$ so in line 10 or 22 $(|\hat{\varsigma}|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ will be propagated.

$\hat{\varsigma}$ is a $\widehat{CApply}$ but not a final state:
Then, $\hat{\varsigma} = (\mathit{clam}, \hat{d}, \mathit{st}, h)$ and $\hat{\varsigma}' = (\llbracket (q\ e)^{\gamma'} \rrbracket, \mathit{st}', h)$.
When $\mathit{Lam}_?(q)$, $\hat{\varsigma}'$ is an inner $\widehat{CEval}$. This case is simple.
When $\mathit{Var}_?(q)$, $\hat{\varsigma}'$ is an Exit-Ret.
By lemma 21, $CE_p(\hat{\varsigma})$ is defined. By the third item of def. 11, p has the form $\hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}_2 \rightsquigarrow \hat{\varsigma}_3 \rightsquigarrow^+ \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$ where $\hat{\varsigma}_2$ is a call, $\hat{\varsigma}_1 = CE_p(\hat{\varsigma}_2) = CE_p(\hat{\varsigma})$ and $\hat{\varsigma}_3 \in CE^*_p(\hat{\varsigma}')$. We must show that $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}|_{\text{al}}) \in \mathit{Seen}$.
Let st_1 be the stack of $\hat{\varsigma}_1$.
The state $\hat{\varsigma}_2$ is a call of the form $(\llbracket (f_2\ e_2\ \mathit{clam}_2)^{l_2} \rrbracket, \mathit{st}_2, h_2)$.
By lemma 20, $\mathit{st}_2 = \mathit{tf}_2 :: \mathit{st}_1$.
By rule $\widehat{[\text{UEA}]}$, $\hat{\varsigma}_3 = (\mathit{ulam}, \hat{d}_3, \mathit{clam}_2, \mathit{st}_3, h_2)$, where $\mathit{st}_3 = \mathit{tf}_3 :: \mathit{st}_1$ and

$$\mathit{tf}_3 = \begin{cases} \mathit{tf}_2 & \mathit{Lam}_?(f_2) \vee H_?(l_2, f_2) \\ \mathit{tf}_2[f_2 \mapsto \{\mathit{ulam}\}] & S_?(l_2, f_2) \end{cases}$$

By lemma 20 for $\hat{\varsigma}_3 \rightsquigarrow^+ \hat{\varsigma}'$, we get $\mathit{st}' = \mathit{tf}' :: \mathit{st}_3$ and $\mathit{tf}'(q) = \mathit{clam}_2$.
Then, by rule $\widehat{[\text{CEA}]}$, $\mathit{clam} = \mathit{clam}_2$, $\mathit{st} = \mathit{st}_3$ and $\hat{d} = \hat{\mathcal{A}}_{\text{u}}(e, \gamma', \mathit{st}', h)$.
The above information will become useful when dealing with the local counterparts of the aforementioned states.
By *IH*, $(|\hat{\varsigma}_3|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$ was entered in W (at line 27) and later examined at line 14. Note that $\hat{\varsigma}_3 \neq \hat{\mathcal{I}}(pr)$ because $\hat{\varsigma}_2$ is between them, therefore `Final` will not be called at line 15.

Also by *IH*, $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}})$ was entered in W and later examined. Lemma 23 implies that $|\hat{\varsigma}_3|_{\text{al}} \in succ(|\hat{\varsigma}_2|_{\text{al}})$ so $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}}, |\hat{\varsigma}_3|_{\text{al}})$ will go in $Callers$. We take cases on whether $(|\hat{\varsigma}_3|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$ or $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}})$ was examined first by the algorithm.

- $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}})$ was examined first
  Then, when $(|\hat{\varsigma}_3|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$ is examined, $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}}, |\hat{\varsigma}_3|_{\text{al}})$ is in $Callers$.
  Therefore, at line 18 we call `Update`$(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}}, |\hat{\varsigma}_3|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$.
  By applying $|\cdot|_{\text{al}}$ to the abstract states we get
  $|\hat{\varsigma}_2|_{\text{al}} = (\llbracket (f_2\ e_2\ clam_2)^{l_2} \rrbracket, tf_2, h_2)$
  $|\hat{\varsigma}_3|_{\text{al}} = (ulam, \hat{d}_3, h_2)$
  $|\hat{\varsigma}'|_{\text{al}} = (\llbracket (q\ e)^{\gamma'} \rrbracket, tf', h)$, where $tf'(q) = clam$.
  From `Update`'s code, we see that the return value is $\tilde{\mathcal{A}}_{\text{u}}(e, \gamma', tf', h) = \hat{\mathcal{A}}_{\text{u}}(e, \gamma', st', h) = \hat{d}$. The frame of the return state is
  $$\begin{cases} tf_2 & Lam_?(f_2) \vee H_?(l_2, f_2) \\ tf_2[f_2 \mapsto \{ulam\}] & S_?(l_2, f_2) \end{cases}$$
  which is equal to tf_3. The heap at the return state is h. Last, the continuation we are returning to is $clam$. Thus, the return state $\tilde{\varsigma}$ is equal to $|\hat{\varsigma}|_{\text{al}}$, and we call `Propagate`$(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$, so $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ will go in $Seen$.
- $(|\hat{\varsigma}_3|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$ was examined first
  Then, when $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}})$ is examined, $(|\hat{\varsigma}_3|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$ is in $Summary$, and at line 12 we call `Update`$(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}}, |\hat{\varsigma}_3|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$. Proceed as above.

$\hat{\varsigma}$ is a final state:

Then, $\hat{\varsigma} = (halt, \hat{d}, \langle\rangle, h)$. We must show that $|\hat{\varsigma}|_{\text{al}}$ will be in $Final$ after the algorithm finishes. By rule $\widehat{[\text{CEA}]}$, $\hat{\varsigma}' = (\llbracket (k\ e)^{\gamma} \rrbracket, st', h)$, where $st' = \langle tf' \rangle$, $tf'(k) = halt$, and $\hat{d} = \hat{\mathcal{A}}_{\text{u}}(e, \gamma, st', h)$.
By lemma 21, $CE^*_p(\hat{\varsigma}') \neq \emptyset$. We will show that $\hat{\mathcal{I}}(pr) \in CE^*_p(\hat{\varsigma}')$.
(We use the same trick as in the proof of lemma 21.)
Assume that $\hat{\mathcal{I}}(pr) \notin CE^*_p(\hat{\varsigma}')$. Let $\hat{\varsigma}_2$ be the earliest state in $CE^*_p(\hat{\varsigma}')$. Then, the path is $\hat{\mathcal{I}}(pr) \rightsquigarrow^+ \hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^+ \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$ where $\hat{\varsigma}_1$ is a call. By lemma 20 for $\hat{\varsigma}_2 \rightsquigarrow^+ \hat{\varsigma}'$, the stack of $\hat{\varsigma}_2$ is empty, so by rule $\widehat{[\text{UEA}]}$, the stack of $\hat{\varsigma}_1$ is empty. But this is impossible by cor. 22. Thus, $\hat{\mathcal{I}}(pr) \in CE^*_p(\hat{\varsigma}')$.
By *IH* for $\hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}'$, we know that $(|\hat{\mathcal{I}}(pr)|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$ was entered in W and $Summary$ at some point. When it was examined, the test at line 14 was true so we called `Final`$(|\hat{\varsigma}'|_{\text{al}})$. Hence, we insert $\tilde{\varsigma} = (halt, \tilde{\mathcal{A}}_{\text{u}}(e, \gamma, tf', h), \emptyset, h)$ in $Final$. But, $\tilde{\mathcal{A}}_{\text{u}}(e, \gamma, tf', h) = \hat{\mathcal{A}}_{\text{u}}(e, \gamma, st', h) = \hat{d}$, hence $\tilde{\varsigma} = |\hat{\varsigma}|_{\text{al}}$.

$\hat{\varsigma}$ is an Exit-Ret:

By lemma 21 for $\hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}'$, p has the form $\hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow^* \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$, where $\hat{\varsigma}_1 = CE_p(\hat{\varsigma}')$. But $\hat{\varsigma}'$ is an $\widehat{Apply}$ state, so by item 2 of def. 11 we get $\hat{\varsigma}_1 = CE_p(\hat{\varsigma})$. By *IH*, $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$ is entered in $Seen$ and W, and examined at line 6. By lemma 23, $|\hat{\varsigma}|_{\text{al}} \in succ(|\hat{\varsigma}'|_{\text{al}})$ so $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ will be propagated (line 7) and entered in $Seen$ (line 27).

We need to show that for every $\hat{\varsigma}'' \in CE_p^*(\hat{\varsigma})$, $(|\hat{\varsigma}''|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ will go in $Seen$.

If $\hat{\varsigma}_1 = \hat{\mathcal{I}}(pr)$ or the predecessor of $\hat{\varsigma}_1$ is a call, then $\hat{\varsigma}_1$ is the only state in $CE_p^*(\hat{\varsigma})$, so we are done.

Otherwise, p has the form $\hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}'' \rightsquigarrow^+ \hat{\varsigma}_2 \rightsquigarrow \hat{\varsigma}_1 \rightsquigarrow^* \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$, where $\hat{\varsigma}_2$ is a tail call and $\hat{\varsigma}'' \in CE_p^*(\hat{\varsigma}_2)$. The subpath $\hat{\varsigma}'' \rightsquigarrow^+ \hat{\varsigma}_2$ has the form $\hat{\varsigma}'' \rightsquigarrow^+ c_1 \rightsquigarrow e_1 \rightsquigarrow^+ c_2 \rightsquigarrow e_2 \rightsquigarrow^+ \ldots \rightsquigarrow^+ c_n \rightsquigarrow e_n \rightsquigarrow^+ \hat{\varsigma}_2$, where $n \geqslant 0$, c_is are tail calls, e_is are entries and $e_i = CE_p(c_{i+1})$.

When $n = 0$, $CE_p(\hat{\varsigma}_2) = \hat{\varsigma}''$. When $n > 0$, $CE_p(\hat{\varsigma}_2) = e_n$ and $CE_p(c_1) = \hat{\varsigma}''$. We assume that $n > 0$ and show $(|\hat{\varsigma}''|_{\text{al}}, |\hat{\varsigma}|_{\text{al}}) \in Seen$.

(For $n = 0$, the proof is simpler.)

First, we show that $(|e_n|_{\text{al}}, |\hat{\varsigma}|_{\text{al}}) \in Seen$; we take cases depending on whether $(|e_n|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}})$ or $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ was processed first.

- $(|e_n|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}})$ was first
  By lemma 23, the triple $(|e_n|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}}, |\hat{\varsigma}_1|_{\text{al}})$ goes in $TCallers$ in line 23. Then, when $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ is examined, we propagate $(|e_n|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ in line 19.
- $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ was first
  In line 17, $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ will go in $Summary$. When $(|e_n|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}})$ is examined, $|\hat{\varsigma}_1|_{\text{al}} \in succ(|\hat{\varsigma}_2|_{\text{al}})$ so in line 24 we propagate $(|e_n|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$.

By repeating this process n times, we can show that, eventually, for any i the edge $(|e_i|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ is put in $Seen$, and the edge $(|\hat{\varsigma}''|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ also goes in $Seen$.

$\hat{\varsigma}$ is none of the above:

$\hat{\varsigma}$ is an inner $\widehat{CEval}$, a call or a tail call. This case is simple. □

The following lemma states that, on an $\widehat{Eval}$-to-$\widehat{Apply}$ transition, the stack below the top frame is irrelevant.

Lemma 24.

- If $(\llbracket (f\ e\ clam)^l \rrbracket, tf :: st, h) \rightsquigarrow (ulam, \hat{d}, clam, tf' :: st, h)$ then for any st', $(\llbracket (f\ e\ clam)^l \rrbracket, tf :: st', h) \rightsquigarrow (ulam, \hat{d}, clam, tf' :: st', h)$

- If $([\![(f\ e\ k)^l]\!], tf :: st, h) \rightsquigarrow (ulam, \hat{d}, \hat{c}, st, h)$ then for any st', $([\![(f\ e\ k)^l]\!], tf :: st', h) \rightsquigarrow (ulam, \hat{d}, \hat{c}, st', h)$
- Similarly for rule $[\widehat{\text{CEA}}]$.

The next lemma states that, on an $\widehat{Apply}$-to-$\widehat{Eval}$ transition, the stack is irrelevant.

Lemma 25.

- If $([\![(\lambda_l (u\ k)\ \ call)]\!], \hat{d}, \hat{c}, st, h) \rightsquigarrow (call, [u \mapsto \hat{d},\ k \mapsto \hat{c}\,] :: st, h')$ then for any st', $([\![(\lambda_l (u\ k)\ \ call)]\!], \hat{d}, \hat{c}, st', h) \rightsquigarrow (call, [u \mapsto \hat{d},\ k \mapsto \hat{c}\,] :: st', h')$
- Similarly for rule $[\widehat{\text{CAE}}]$, where st' is any non-empty stack.

When explaining the intuition behind summarization in section 4.3.1, we mentioned that the return point of a function does not influence reachability while we are computing inside the function. The next lemma formalizes that insight. In push monotonic paths, we can replace the stack of the first state with an arbitrary stack and get a path of the same structure.

Lemma 26 (Stack irrelevance)**.** Let $p = \hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow \ldots \rightsquigarrow \hat{\varsigma}_n$ be push monotonic, where $\hat{\varsigma}_1 = (ulam, \hat{d}, \hat{c}, st_1, h_1)$ and $\hat{\varsigma}_n$ is not a $\widehat{CApply}$ of the form $(\hat{c}, \ldots, st_1, \ldots)$. The stack of each $\hat{\varsigma}_i$ is of the form $st_i st_1$.
For an arbitrary stack st' and continuation $\hat{c}'$, consider the sequence p' of states $\hat{\varsigma}'_1\ \hat{\varsigma}'_2\ \ldots\ \hat{\varsigma}'_n$ where each $\hat{\varsigma}'_i$ is produced by $\hat{\varsigma}_i$ as follows:

- if $\hat{\varsigma}_i$ has the form $(ulam_i, \hat{d}_i, \hat{c}, st_1, h_i)$, change $\hat{c}$ to $\hat{c}'$ and st_1 to st'.
- if st_1 is a proper suffix of the stack of $\hat{\varsigma}_i$, the latter has the form $st_i\langle fr_i\rangle st_1$. Change st_1 to st' and bind the continuation variable in fr_i to $\hat{c}'$.

(The map is not total, but it should be defined for all states in p.)
Then,

- for any two states $\hat{\varsigma}'_i$ and $\hat{\varsigma}'_{i+1}$ in p', it holds that $\hat{\varsigma}'_i \rightsquigarrow \hat{\varsigma}'_{i+1}$
- the path p' is push monotonic
- the path structure is preserved, *i.e.*, CE_p and CE_p^* relations between states transfer over to the new path.

Proof. The proof is by induction on the length of p.
The base case is simple.
When the length is greater than 0, p is $\hat{\varsigma}_1 \rightsquigarrow^* \hat{\varsigma}_{n-1} \rightsquigarrow \hat{\varsigma}_n$. By *IH*, the transitions

in $\hat{\varsigma}'_1 \rightsquigarrow^* \hat{\varsigma}'_{n-1}$ are valid with respect to the abstract semantics, $\hat{\varsigma}'_1 \rightsquigarrow^* \hat{\varsigma}'_{n-1}$ is push monotonic and the path structure is preserved. We must show:

- $(\hat{\varsigma}'_{n-1}, \hat{\varsigma}'_n) \in \rightsquigarrow$
- $\hat{\varsigma}'_1 \rightsquigarrow^* \hat{\varsigma}'_n$ is push monotonic.
- if $\hat{\varsigma}_j = CE_p(\hat{\varsigma}_n)$ then $\hat{\varsigma}'_j = CE_p(\hat{\varsigma}'_n)$.
- if $\hat{\varsigma}_j \in CE^*_p(\hat{\varsigma}_n)$ then $\hat{\varsigma}'_j \in CE^*_p(\hat{\varsigma}'_n)$.

We take cases on $\hat{\varsigma}_{n-1}$.

$\hat{\varsigma}_{n-1}$ ***is a*** $\widehat{UEval}$***:***
$\hat{\varsigma}_{n-1}$ has the form $([\![(f\ e\ q)^l]\!], st, h)$. By lemma 21, $CE^*_p(\hat{\varsigma}_{n-1}) \neq \emptyset$.
If $\hat{\varsigma}_1 \in CE^*_p(\hat{\varsigma}_{n-1})$ then by lemma 20, st has the form $tf :: st_1$ where $tf(k) = \hat{c}$ for some variable k.

- q is a variable, so $q = k$
  By rule $[\widehat{\text{UEA}}]$, $\hat{\varsigma}_n$ is $(ulam_n, \hat{d}_n, \hat{c}, st_1, h)$. With the new continuation $\hat{c}'$ and stack st', the state $\hat{\varsigma}'_{n-1}$ is $([\![(f\ e\ q)^l]\!], tf[q \mapsto \hat{c}'] :: st', h)$, and it transitions to $(ulam_n, \hat{d}_n, \hat{c}', st', h)$, which is $\hat{\varsigma}'_n$.
  Push monotonicity is preserved by this transition.
- q is a lambda and f is a stack reference
  By rule $[\widehat{\text{UEA}}]$, $\hat{\varsigma}_n$ is $(ulam_n, \hat{d}_n, q, tf[f \mapsto \{ulam_n\}] :: st_1, h)$.
  With $\hat{c}'$ and st', the state $\hat{\varsigma}'_{n-1}$ is $([\![(f\ e\ q)^l]\!], tf[k \mapsto \hat{c}'] :: st', h)$, and its successor is $(ulam_n, \hat{d}_n, q, tf[k \mapsto \hat{c}',\ f \mapsto \{ulam_n\}] :: st', h)$, which is $\hat{\varsigma}'_n$.
  Again, push monotonicity is preserved.
- q is a lambda and f is a heap reference
  Similarly.

If $\hat{\varsigma}_1 \notin CE^*_p(\hat{\varsigma}_{n-1})$ then by lemmas 20 and 21, we can show that st has at least two more frames than st_1. Thus, lemma 24 gives us $\hat{\varsigma}'_{n-1} \rightsquigarrow \hat{\varsigma}'_n$.
When $\hat{\varsigma}_{n-1}$ is a call, $\hat{\varsigma}'_n$ obviously preserves path structure.
When $\hat{\varsigma}_{n-1}$ is a tail call, $\hat{\varsigma}'_{n-1}$ is also a tail call.
We know $CE^*_p(\hat{\varsigma}_n) = \{\hat{\varsigma}_n\} \cup CE^*_p(\hat{\varsigma}_{n-1})$ and $CE^*_p(\hat{\varsigma}'_n) = \{\hat{\varsigma}'_n\} \cup CE^*_p(\hat{\varsigma}'_{n-1})$.
Thus, it suffices to show that if $\hat{\varsigma}_j \in CE^*_p(\hat{\varsigma}_{n-1})$ then $\hat{\varsigma}'_j \in CE^*_p(\hat{\varsigma}'_{n-1})$. But that is true by *IH* for $\hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}_{n-1}$.

$\hat{\varsigma}_{n-1}$ ***is an Exit-Ret:***
By lemma 21, $CE^*_p(\hat{\varsigma}_{n-1}) \neq \emptyset$.
If $\hat{\varsigma}_1 \in CE^*_p(\hat{\varsigma}_{n-1})$, then by lemma 20 $\hat{\varsigma}_n$ has the form $(\hat{c}, \ldots, st_1, \ldots)$, which

we do not allow. Thus, $\hat{\varsigma}_1 \notin CE^*_p(\hat{\varsigma}_{n-1})$. Let $\hat{\varsigma}_e$ be the earliest state in $CE^*_p(\hat{\varsigma}_{n-1})$; its predecessor $\hat{\varsigma}_c$ is a call.
Using lemmas 20 and 21 we can show that st has at least two more frames than st_1. Lemma 24 gives us $\hat{\varsigma}'_{n-1} \rightsquigarrow \hat{\varsigma}'$.
By the third item of def. 11, $CE_p(\hat{\varsigma}_n) = CE_p(\hat{\varsigma}_c)$. But by *IH*, $\hat{\varsigma}'_e \in CE^*_p(\hat{\varsigma}_{n-1})$, so def. 11 gives us $CE_p(\hat{\varsigma}'_n) = CE_p(\hat{\varsigma}'_c)$.
Last, we must show that if $\hat{\varsigma}_j \in CE^*_p(\hat{\varsigma}_n)$ then $\hat{\varsigma}'_j \in CE^*_p(\hat{\varsigma}'_n)$.
Clearly, $CE^*_p(\hat{\varsigma}_n) = CE^*_p(\hat{\varsigma}_c)$ and $CE^*_p(\hat{\varsigma}'_n) = CE^*_p(\hat{\varsigma}'_c)$. Thus, it suffices to show that if $\hat{\varsigma}_j \in CE^*_p(\hat{\varsigma}_c)$ then $\hat{\varsigma}'_j \in CE^*_p(\hat{\varsigma}'_c)$, which holds by *IH*.

$\hat{\varsigma}_{n-1}$ ***is an entry:***
Then, $CE^*_p(\hat{\varsigma}_{n-1}) \neq \emptyset$. If $\hat{\varsigma}_1 \notin CE^*_p(\hat{\varsigma}_{n-1})$ then by lemmas 20 and 21 we can show that the stack of $\hat{\varsigma}_{n-1}$ has at least one more frame than st_1.
By lemma 25, we get $\hat{\varsigma}'_{n-1} \rightsquigarrow \hat{\varsigma}'_n$.
When $\hat{\varsigma}_1 \in CE^*_p(\hat{\varsigma}_{n-1})$, the proof is similar to the previous cases.

$\hat{\varsigma}_{n-1}$ ***is none of the above:***
Then, $\hat{\varsigma}_{n-1}$ is a $\widehat{CApply}$ or an inner $\widehat{CEval}$. Similarly. □

Lemma 27. If $\tilde{\varsigma} \approx\!\!> \tilde{\varsigma}'$ then, for any $\hat{\varsigma}$ such that $\tilde{\varsigma} = |\hat{\varsigma}|_{\text{al}}$, there exists a state $\hat{\varsigma}'$ such that $\hat{\varsigma} \rightsquigarrow \hat{\varsigma}'$ and $\tilde{\varsigma}' = |\hat{\varsigma}'|_{\text{al}}$.

Theorem (Completeness)**.** After summarization:

- For each $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ in $Seen$, there exist states $\hat{\varsigma}_1, \hat{\varsigma}_2 \in \mathcal{RS}$ such that $\hat{\varsigma}_1 \rightsquigarrow^* \hat{\varsigma}_2$ and $\tilde{\varsigma}_1 = |\hat{\varsigma}_1|_{\text{al}}$ and $\tilde{\varsigma}_2 = |\hat{\varsigma}_2|_{\text{al}}$ and $\hat{\varsigma}_1 \in CE^*_p(\hat{\varsigma}_2)$
- For each $\tilde{\varsigma}$ in $Final$, there exists a final state $\hat{\varsigma} \in \mathcal{RS}$ such that $\tilde{\varsigma} = |\hat{\varsigma}|_{\text{al}}$

Proof. By induction on the number of iterations. We prove that the algorithm maintains the following properties for $Seen$ and $Final$.

1. For each $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ in $Seen$, there exist states $\hat{\varsigma}_1, \hat{\varsigma}_2 \in \mathcal{RS}$ and path p such that $p = \hat{\varsigma}_1 \rightsquigarrow^* \hat{\varsigma}_2$ and $\tilde{\varsigma}_1 = |\hat{\varsigma}_1|_{\text{al}}$ and $\tilde{\varsigma}_2 = |\hat{\varsigma}_2|_{\text{al}}$ and, if $\tilde{\varsigma}_2$ is an Exit-Ret then $\hat{\varsigma}_1 \in CE^*_p(\hat{\varsigma}_2)$ otherwise $\hat{\varsigma}_1 = CE_p(\hat{\varsigma}_2)$
2. For each $\tilde{\varsigma}$ in $Final$, there exists a final state $\hat{\varsigma} \in \mathcal{RS}$ such that $\tilde{\varsigma} = |\hat{\varsigma}|_{\text{al}}$

Initially, we show that the properties hold before the first iteration (at the beginning of the algorithm): $Final$ is empty and W contains just $(\tilde{\mathcal{I}}(pr), \tilde{\mathcal{I}}(pr))$, for which property (1) holds.

Now the inductive step: at the beginning of each iteration, we remove an edge $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ from W. We assume that the properties hold at that point. We must show that, after we process the edge, the new elements of $Seen$ and $Final$ satisfy the properties. We take cases on $\hat{\varsigma}_2$.

$\tilde{\varsigma}_2$ is an entry, a $\widetilde{CApply}$ or an inner $\widetilde{CEval}$:
$(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ is in $Seen$, so by *IH*
$\exists\, \hat{\varsigma}_1, \hat{\varsigma}_2 \in \mathcal{RS}.\ p = \hat{\varsigma}_1 \rightsquigarrow^* \hat{\varsigma}_2 \ \wedge\ \tilde{\varsigma}_1 = |\hat{\varsigma}_1|_{\text{al}} \ \wedge\ \tilde{\varsigma}_2 = |\hat{\varsigma}_2|_{\text{al}} \ \wedge\ \hat{\varsigma}_1 = CE_p(\hat{\varsigma}_2)$
For each $\tilde{\varsigma}_3$ in $succ(\tilde{\varsigma}_2)$, $(\tilde{\varsigma}_1, \tilde{\varsigma}_3)$ will be propagated.
If $(\tilde{\varsigma}_1, \tilde{\varsigma}_3)$ is already in $Seen$, (1) holds by *IH*. (From now on, we will not repeat this argument; we will assume that the insertion in $Seen$ happens now.)
Otherwise, we insert the edge at this iteration, at line 27. By lemma 27, $\exists\, \hat{\varsigma}_3.\ \tilde{\varsigma}_3 = |\hat{\varsigma}_3|_{\text{al}} \ \wedge\ \hat{\varsigma}_2 \rightsquigarrow \hat{\varsigma}_3$. By item 2 of def. 11, $\hat{\varsigma}_1 = CE_p(\hat{\varsigma}_3)$.

$\tilde{\varsigma}_2$ is a call:
Let $\tilde{\varsigma}_1 = (ulam_1, \hat{d}_1, h_1)$ and $\tilde{\varsigma}_2 = (\llbracket (f\ e\ clam)^{l_2} \rrbracket, tf_2, h_2)$.
Also, assume $S_?(l_2, f)$ (the other cases are simpler).
$(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ is in $Seen$, so by *IH*
$\exists\, \hat{\varsigma}_1, \hat{\varsigma}_2 \in \mathcal{RS}.\ p = \hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}_2 \ \wedge\ \tilde{\varsigma}_1 = |\hat{\varsigma}_1|_{\text{al}} \ \wedge\ \tilde{\varsigma}_2 = |\hat{\varsigma}_2|_{\text{al}} \ \wedge\ \hat{\varsigma}_1 = CE_p(\hat{\varsigma}_2)$
Each entry $\tilde{\varsigma}_3$ in $succ(\tilde{\varsigma}_2)$ will be propagated.
By lemma 27, $\exists\, \hat{\varsigma}_3.\ \tilde{\varsigma}_3 = |\hat{\varsigma}_3|_{\text{al}} \ \wedge\ \hat{\varsigma}_2 \rightsquigarrow \hat{\varsigma}_3$. Since $\hat{\varsigma}_3 = CE_p(\hat{\varsigma}_3)$, (1) holds for $\tilde{\varsigma}_3$.
If there is no edge $(\tilde{\varsigma}_3, \tilde{\varsigma}_4)$ in $Summary$, we are done. Otherwise, we must show that (1) holds for the edge inserted in $Seen$ by `Update`$(\tilde{\varsigma}_1, \tilde{\varsigma}_2, \tilde{\varsigma}_3, \tilde{\varsigma}_4)$.
Let st_1 be the stack of $\hat{\varsigma}_1$. By lemma 20, the stack of $\hat{\varsigma}_2$ is $tf_2 :: st_1$.
Let $\tilde{\varsigma}_3 = (ulam_3, \hat{d}_3, h_2)$ and $\tilde{\varsigma}_4 = (\llbracket (k_4\ e_4)^{l_4} \rrbracket, tf_4, h_4)$.
(Note that tf_4 contains only user bindings.) We know $Summary \subseteq Seen$ so by *IH* for $(\tilde{\varsigma}_3, \tilde{\varsigma}_4)$ we get (note that $\tilde{\varsigma}_4$ is an Exit-Ret)
$\exists\, \hat{\varsigma}'_3, \hat{\varsigma}'_4 \in \mathcal{RS}.\ p' = \hat{\varsigma}'_3 \rightsquigarrow^+ \hat{\varsigma}'_4 \ \wedge\ \tilde{\varsigma}_3 = |\hat{\varsigma}'_3|_{\text{al}} \ \wedge\ \tilde{\varsigma}_4 = |\hat{\varsigma}'_4|_{\text{al}} \ \wedge\ \hat{\varsigma}'_3 \in CE^*_{p'}(\hat{\varsigma}'_4)$
Then, $\hat{\varsigma}'_3 = (ulam_3, \hat{d}_3, \hat{c}_3, st'_3, h_2)$ and by lemma 20,
$\hat{\varsigma}'_4 = (\llbracket (k_4\ e_4)^{l_4} \rrbracket, tf_4[k_4 \mapsto \hat{c}_3\,] :: st'_3, h_4)$.
But p' is push monotonic, so by lemma 26 there exist states
$\hat{\varsigma}_3 = (ulam_3, \hat{d}_3, clam, st_3, h_2)$ where $st_3 = tf_2[f \mapsto \{ulam_3\}\,] :: st_1$ and
$\hat{\varsigma}_4 = (\llbracket (k_4\ e_4)^{l_4} \rrbracket, st_4, h_4)$ where $st_4 = tf_4[k_4 \mapsto clam\,] :: st_3$
such that $\hat{\varsigma}_3 \rightsquigarrow^+ \hat{\varsigma}_4$ and $\hat{\varsigma}_3 \in CE^*_p(\hat{\varsigma}_4)$.
Now, we can extend p to $\hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}_2 \rightsquigarrow \hat{\varsigma}_3 \rightsquigarrow^+ \hat{\varsigma}_4$.
By rule $[\widehat{\text{CEA}}]$, the successor $\hat{\varsigma}$ of $\hat{\varsigma}_4$ is $(clam, \hat{\mathcal{A}}_{\text{u}}(e_4, l_4, st_4, h_4), st_3, h_4)$.
The state $\tilde{\varsigma}$ produced by `Update` is $(clam, \tilde{\mathcal{A}}_{\text{u}}(e_4, l_4, tf_4, h_4), tf, h_4)$ where

$tf = tf_2[f \mapsto \{ulam_3\}]$. It is simple to see that $\tilde{\varsigma} = |\hat{\varsigma}|_{\text{al}}$.
Also, by item 3 of def. 11, $\hat{\varsigma}_1 = CE_p(\hat{\varsigma})$.

$\tilde{\varsigma}_2$ is an Exit-Ret:
$\tilde{\varsigma}_2$ has the form $([\![(k\ e)^\gamma]\!], tf, h)$.
If $\tilde{\varsigma}_1$ is $\tilde{\mathcal{I}}(pr)$ then we call `Final`$(\tilde{\varsigma}_2)$ and we insert a local state $\tilde{\varsigma}$ of the form $(halt, \tilde{\mathcal{A}}_{\text{u}}(e, \gamma, tf, h), \emptyset, h)$ in $Final$. We must show that (2) holds.
By *IH* for $(\tilde{\mathcal{I}}(pr), \tilde{\varsigma}_2)$, $\exists\, \hat{\varsigma}_2.\ p = \hat{\mathcal{I}}(pr) \rightsquigarrow^+ \hat{\varsigma}_2 \ \wedge\ \tilde{\varsigma}_2 = |\hat{\varsigma}_2|_{\text{al}} \ \wedge\ \hat{\mathcal{I}}(pr) \in CE_p^*(\hat{\varsigma}_2)$.
By lemma 20, the stack st of $\hat{\varsigma}_2$ is $\langle tf[k \mapsto halt]\rangle$. Hence, the successor $\hat{\varsigma}$ of $\hat{\varsigma}_2$ is $(halt, \hat{\mathcal{A}}_{\text{u}}(e, \gamma, st, h), \langle\rangle, h)$, and $\tilde{\varsigma} = |\hat{\varsigma}|_{\text{al}}$ holds.
If $\tilde{\varsigma}_1 \neq \tilde{\mathcal{I}}(pr)$, for each triple $(\tilde{\varsigma}_3, \tilde{\varsigma}_4, \tilde{\varsigma}_1)$ in $Callers$, we call `Update`$(\tilde{\varsigma}_3, \tilde{\varsigma}_4, \tilde{\varsigma}_1, \tilde{\varsigma}_2)$.
We can work as in the previous case to show that the edge inserted in $Seen$ by `Update` satisfies (1).
For each triple $(\tilde{\varsigma}_3, \tilde{\varsigma}_4, \tilde{\varsigma}_1)$ in $TCallers$, we propagate $(\tilde{\varsigma}_3, \tilde{\varsigma}_2)$ (line 19). We must show that $(\tilde{\varsigma}_3, \tilde{\varsigma}_2)$ satisfies (1). Insertion in $TCallers$ happens only at line 23, which means that $(\tilde{\varsigma}_3, \tilde{\varsigma}_4)$ is in $Seen$. Thus, by *IH*
$\exists\, \hat{\varsigma}_3, \hat{\varsigma}_4 \in \mathcal{RS}.\ p = \hat{\varsigma}_3 \rightsquigarrow^+ \hat{\varsigma}_4 \ \wedge\ \tilde{\varsigma}_3 = |\hat{\varsigma}_3|_{\text{al}} \ \wedge\ \tilde{\varsigma}_4 = |\hat{\varsigma}_4|_{\text{al}} \ \wedge\ \hat{\varsigma}_3 = CE_p(\hat{\varsigma}_4)$
Also, $\tilde{\varsigma}_4 \not\twoheadrightarrow \tilde{\varsigma}_1$ so by lemma 27 $\exists\, \hat{\varsigma}_1.\ \hat{\varsigma}_4 \rightsquigarrow \hat{\varsigma}_1 \ \wedge\ \tilde{\varsigma}_1 = |\hat{\varsigma}_1|_{\text{al}}$. By *IH* for $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$
$\exists\, \hat{\varsigma}'_1, \hat{\varsigma}'_2 \in \mathcal{RS}.\ p' = \hat{\varsigma}'_1 \rightsquigarrow^+ \hat{\varsigma}'_2 \ \wedge\ \tilde{\varsigma}_1 = |\hat{\varsigma}'_1|_{\text{al}} \ \wedge\ \tilde{\varsigma}_2 = |\hat{\varsigma}'_2|_{\text{al}} \ \wedge\ \hat{\varsigma}'_1 = CE_{p'}^*(\hat{\varsigma}'_2)$
We can stitch these three together using lemma 26, so we can grow p to $\hat{\varsigma}_3 \rightsquigarrow^+ \hat{\varsigma}_4 \rightsquigarrow \hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}_2$, where $\hat{\varsigma}_3 \in CE_p^*(\hat{\varsigma}_2)$.

$\tilde{\varsigma}_2$ is a tail call:
Similarly. □

A.2 Proofs for CFA2 with first-class control

These proofs use the semantics without stack filtering for simplicity. Stack filtering is orthogonal to first-class control; one can easily adapt the reasoning about stack filtering from the proofs in the previous section to the proofs in this section.

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

Proof. By cases on the concrete transition. We only show the theorem for the transitions that involve first-class control, *i.e.*, rules $[\widehat{\text{UAE}}]$ and $[\widehat{\text{CEA}}]$.

Rule $[\text{UAE}]$***:***
$(\langle [\![(\lambda_l (u\ k)\ call)]\!], \rho \rangle, d, c, ve, t) \to (call, \rho', ve', t)$
$\rho' = \rho[u \mapsto t,\ k \mapsto t\,]$
$ve' = ve[(u,t) \mapsto d,\ (k,t) \mapsto c\,]$

Let $ts = \begin{cases} \langle\rangle & c = halt \\ toStack(LV(\mathcal{L}(lam)), \rho_1, ve) & c = (lam, \rho_1) \end{cases}$
Since $|\varsigma|_{\text{ca}} \sqsubseteq \hat{\varsigma}$, $\hat{\varsigma}$ is of the form $([\![(\lambda_l (u\ k)\ call)]\!], \hat{d}, \hat{c}, st, h)$,
where $|d|_{\text{ca}} \sqsubseteq \hat{d}$, $|c|_{\text{ca}} = \hat{c}$, $ts \sqsubseteq st$ and $|ve|_{\text{ca}} \sqsubseteq h$.

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

Let ts' be the stack of $|\varsigma'|_{\text{ca}}$.
From $iu_\lambda(call) = \lambda_l$ we get $ts' = toStack(LV(l), \rho', ve')$.
We must show that $|\varsigma'|_{\text{ca}} \sqsubseteq \hat{\varsigma}'$, *i.e.*, $ts' \sqsubseteq st'$ and $|ve'|_{\text{ca}} \sqsubseteq h'$.

Showing $ts' \sqsubseteq st'$ is not impacted by first-class control, so the previous proof does not change.
If $H_?(k)$, then we must show $|ve'|_{\text{ca}}(k) \sqsubseteq h'(k)$. We assume that $c = (lam, \rho_1)$. (Showing it for $c = halt$ is similar.)
We know that $|ve'|_{\text{ca}}(k) = |ve|_{\text{ca}}(k) \cup \{\langle lam, toStack(LV(\mathcal{L}(lam)), \rho_1, ve')\rangle\}$

and $h'(k) = h(k) \cup \{\langle lam, st\rangle\}$. Since $|ve|_{\text{ca}} \sqsubseteq h$, it is enough to show $toStack(LV(\mathcal{L}(lam)), \rho_1, ve') \sqsubseteq st$. But we already know that $ts \sqsubseteq st$, so we can just show $ts = toStack(LV(\mathcal{L}(lam)), \rho_1, ve')$. This holds by lemma 6, because the two bindings of ve' born at time t are younger than all bindings in ρ_1.

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

Let $ts = toStack(LV(\gamma), \rho, ve)$. Since $|\varsigma|_{\text{ca}} \sqsubseteq \hat{\varsigma}$, $\hat{\varsigma}$ is of the form $([\![(q\ e)^\gamma]\!], st, h)$, where $|ve|_{\text{ca}} \sqsubseteq h$ and $ts \sqsubseteq st$. The abstract transition is
$([\![(q\ e)^\gamma]\!], st, h) \rightsquigarrow (\hat{c}, \hat{d}, st', h)$
$\hat{d} = \hat{\mathcal{A}}_{\text{u}}(e, \gamma, st, h)$

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

Let ts' be the stack of $|\varsigma'|_{\text{ca}}$. We must show that $|\varsigma'|_{\text{ca}} \sqsubseteq \hat{\varsigma}'$, *i.e.*, $|proc|_{\text{ca}} = \hat{c}$, $|d|_{\text{ca}} \sqsubseteq \hat{d}$, and $ts' \sqsubseteq st'$.
Showing $|d|_{\text{ca}} \sqsubseteq \hat{d}$ is simple, by cases on e.
We show the other two for the first-class-control case, *i.e.*, when $H_?(\gamma, q)$.
We assume that $proc = ve(q, \rho(q)) = (lam, \rho_1)$.
(The proof is similar when $ve(q, \rho(q)) = halt$).
In this case, $|proc|_{\text{ca}} = lam$ and $(lam, toStack(LV(\mathcal{L}(lam)), \rho_1, ve)) \in |ve|_{\text{ca}}(q)$.
Since $|ve|_{\text{ca}} \sqsubseteq h$, there exists a pair $(lam, st') \in h(q)$ such that $toStack(LV(\mathcal{L}(lam)), \rho_1, ve) \sqsubseteq st'$. We pick this pair for $\hat{\varsigma}'$.
But then, $ts' \sqsubseteq st'$ because $ts' = toStack(LV(\mathcal{L}(lam)), \rho_1, ve)$. □

For the proof of the soundness theorem, we need the following lemma. It is a combination of the same-level reachability and path-decomposition lemmas (lemmas 20 and 21), but slightly weaker because it does not make any claims about push monotonicity. (Push monotonicity is trickier to define in the presence of first-class control, but we do not need it to prove soundness.)

Lemma 28 (Path decomposition)**.** Let $p = \hat{\mathcal{I}}(pr) \leadsto^* \hat{\varsigma}$ where $\hat{\varsigma} = (\ldots, st, h)$.

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 $st \neq \langle\rangle$ 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 $\hat{\varsigma}_e = (\llbracket (\lambda_l (u\ k)\ call) \rrbracket, \hat{d}, \hat{c}, st_e, h_e)$.
   If $\hat{\varsigma}_e \in (CE_p^*(\hat{\varsigma}) \setminus CE_p(\hat{\varsigma}))$ then

   - there is a frame tf such that $st = tf :: st_e$.
   - 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 $st = tf :: st_e$, $\mathrm{dom}(tf) \subseteq LV(l)$, $tf(u) = \hat{d}$, $tf(k) = \hat{c}$.
   - 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 of p is 0, then $\hat{\varsigma} = \hat{\mathcal{I}}(pr)$ and $CE_p(\hat{\varsigma}) = CE_p^*(\hat{\varsigma}) = \{\hat{\varsigma}\}$. The lemma trivially holds.

If the length is greater than 0, p has the form $\hat{\mathcal{I}}(pr) \leadsto^* \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$. We take cases on $\hat{\varsigma}$. We only show the cases that involve first-class control.

$\hat{\varsigma}$ 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 $st \neq \langle\rangle$. 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 $[\widehat{\mathrm{CAE}}]$, st is also non-empty.

$\hat{\varsigma}$ is a $\widehat{CApply}$ and $\hat{\varsigma}'$ is an Exit-Esc:
Let $\hat{\varsigma} = (\hat{c}, \hat{d}, st, h)$ and $\hat{\varsigma}' = (\llbracket (k\ e)^{\gamma} \rrbracket, st', h)$.
(By *IH*, $st' \neq \langle\rangle$. This is necessary for the transition to occur, because e may be a stack reference.)
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. The path has the form
$\hat{\mathcal{I}}(pr) \rightsquigarrow^+ \hat{\varsigma}_4 \rightsquigarrow \hat{\varsigma}_3 \rightsquigarrow^* \hat{\varsigma}_2 \rightsquigarrow^+ \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$.
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. 16, if $CE_p^*(\hat{\varsigma})$ is not empty, then the path can be decomposed according to the fourth case:
  $\hat{\mathcal{I}}(pr) \rightsquigarrow^+ \hat{\varsigma}_3 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$
  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 $\hat{\mathcal{I}}(pr) \rightsquigarrow^+ \hat{\varsigma}_3 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$.
  By *IH* for $\hat{\mathcal{I}}(pr) \rightsquigarrow^+ \hat{\varsigma}_1$, we get that the continuation argument of $\hat{\varsigma}_2$ is $\hat{c}$ and its stack is st. Then, by rule $\widehat{[\mathrm{UEA}]}$, 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 item 4 of def. 16, $CE_p(\hat{\varsigma}_3) \subseteq CE_p(\hat{\varsigma})$. But by *IH*, 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 $CE_p(\hat{\varsigma}) \cap CE_p(\hat{\varsigma}_3)$. (Without loss of generality, the following arguments apply to all states in $CE_p(\hat{\varsigma})$, because any decomposition of p to find corresponding entries for $\hat{\varsigma}$ has the form we used above.)
Let $\hat{\varsigma}_e \in CE_p(\hat{\varsigma}_3)$, 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*, $st_3 = tf :: st_e$, $\mathrm{dom}(tf) \subseteq LV(l)$, $tf(u) = \hat{d}_e$, $tf(k') = \hat{c}_e, l' \in LL(l)$.
$\hat{\varsigma}_2$ has the form $(ulam, \hat{d}_2, \hat{c}, st, h)$ where $st = st_3$. Thus, st has the appropriate form. Also, $\hat{c}$ is a lambda appearing at l', so $\mathcal{L}(\hat{c}) \in LL(l)$.
When $\hat{\varsigma}_e \in CE^*_p(\hat{\varsigma}_3) \setminus CE_p(\hat{\varsigma}_3)$, the proof is similar and simpler. □

Lemma 29 (Local simulation)**.**
Let $\hat{\varsigma} \in \mathcal{RS}$ and $\hat{\varsigma} \rightsquigarrow \hat{\varsigma}'$. If $\hat{\varsigma}$ is not an Exit-Ret or Exit-Esc, $|\hat{\varsigma}'|_{\mathrm{al}} \in succ(|\hat{\varsigma}|_{\mathrm{al}})$.

Theorem (Soundness)**.** Let $p = \hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}$. Then, after summarization:

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

Proof. By induction on the length of p.
Note that we slightly strengthen theorem 18 by adding one more proof obligation, in order to reason about summaries for escaping continuations.
The base case is simple.
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 $\hat{\varsigma}$. We only show the cases that involve first-class control.

$\hat{\varsigma}$ is a $\widehat{CApply}$ and $\hat{\varsigma}'$ is an Exit-Esc:
Let $\hat{\varsigma} = (\hat{c}, \hat{d}, st, h)$ and $\hat{\varsigma}' = ([\![(k\ e)^{\gamma}]\!], st', h)$. By lemma 28, $CE^*_p(\hat{\varsigma}') \neq \emptyset$.
Since $(\hat{c}, st) \in h(k)$, there exists $\hat{\varsigma}_1$ in $CE_p(\hat{\varsigma}')$ of the form $(def_\lambda(k), \hat{d}_1, \hat{c}, st, h_1)$.
Thus, p can be written $\hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$. 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 28 for $\hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}_1$, we get $\hat{c} = halt$ and $st = \langle\rangle$. Thus, $\hat{\varsigma}$ is a final state. By *IH*, $(|\hat{\mathcal{I}}(pr)|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$ was put in $Summary$ before it was examined. Therefore, when it was examined, the test at line 26 was false.
  The test at line 30 was true, so `Final`$(|\hat{\varsigma}'|_{\text{al}})$ was called. By lemma 28 for $\hat{\mathcal{I}}(pr) \rightsquigarrow^+ \hat{\varsigma}'$, we get that st' is not empty, so it has the form $tf :: st''$. Then, $|\hat{\varsigma}'|_{\text{al}}$ is $([\![(k\ e)^\gamma]\!], tf \restriction UVar, h \restriction UVar)$.
  We put the state $(halt, \tilde{\mathcal{A}}_{\text{u}}(e, \gamma, tf \restriction UVar, h \restriction UVar), \emptyset, h \restriction UVar)$ in $Final$.
  But this state is $|\hat{\varsigma}|_{\text{al}}$ because $\tilde{\mathcal{A}}_{\text{u}}(e, \gamma, tf \restriction UVar, h \restriction UVar)$ is equal to $\hat{\mathcal{A}}_{\text{u}}(e, \gamma, st', h)$.
- $\hat{\mathcal{I}}(pr) \notin CE_p^*(\hat{\varsigma}_1)$
  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,
  $p = \hat{\mathcal{I}}(pr) \rightsquigarrow^+ \hat{\varsigma}_3 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}' \rightsquigarrow \hat{\varsigma}$
  By lemma 28, we find that the continuation argument of $\hat{\varsigma}_2$ is $\hat{c}$ and its stack is st. By rule $[\widehat{\text{UEA}}]$, $\hat{c}$ is the continuation lambda passed at $\hat{\varsigma}_3$. Therefore, $\hat{\varsigma}$ is not a final state. By def. 16 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 $(|\hat{\varsigma}_4|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ was put in $Seen$. By *IH*, we know that $(|\hat{\varsigma}_4|_{\text{al}}, |\hat{\varsigma}_3|_{\text{al}})$ and $(|\hat{\varsigma}_2|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$ were put in W and examined. We take cases on which edge was examined first.
  Assume $(|\hat{\varsigma}_4|_{\text{al}}, |\hat{\varsigma}_3|_{\text{al}})$ was examined first. By lemma 29, $|\hat{\varsigma}_2|_{\text{al}} \in succ(|\hat{\varsigma}_3|_{\text{al}})$, so at line 17 we put $(|\hat{\varsigma}_4|_{\text{al}}, |\hat{\varsigma}_3|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}})$ in $Callers$. We later examine $(|\hat{\varsigma}_2|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$. By *IH*, it is in $Summary$ when it's 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 $(|\hat{\varsigma}_4|_{\text{al}}, |\hat{\varsigma}_3|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}})$ is in $Callers$, we call `Update`$(|\hat{\varsigma}_4|_{\text{al}}, |\hat{\varsigma}_3|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$ at line 32. We must show that the state $\tilde{\varsigma}$ created by `Update` is $|\hat{\varsigma}|_{\text{al}}$.
  By lemma 28, we get that $st' \neq \langle\rangle$. It is easy to see that the user value passed at $\tilde{\varsigma}$, which is $\tilde{\mathcal{A}}_{\text{u}}(e, \gamma, |st'|_{\text{al}}, |h|_{\text{al}})$, is equal to $\hat{\mathcal{A}}_{\text{u}}(e, \gamma, st', h)$.
  By lemma 28, the stack of $\hat{\varsigma}_3$ is not empty.
  Thus, $\hat{\varsigma}_3$ has the form $([\![(e_1\ e_2\ q)^l]\!], tf :: st_3, h_3)$ where $q = \hat{c}$.
  By rule $[\widehat{\text{UEA}}]$, the stack st of $\hat{\varsigma}_2$ is $tf :: st_3$.
  Therefore, the frame of $\tilde{\varsigma}$ is equal to $|st|_{\text{al}}$, so $\tilde{\varsigma} = |\hat{\varsigma}|_{\text{al}}$.
  Assume that $(|\hat{\varsigma}_2|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$ was examined first. Then, when $(|\hat{\varsigma}_4|_{\text{al}}, |\hat{\varsigma}_3|_{\text{al}})$ 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 28, $CE_p(\hat{\varsigma}') \neq \emptyset$. Let $\hat{\varsigma}_1 \in CE_p(\hat{\varsigma}')$. By *IH*, $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}'|_{\text{al}})$ was examined. Also, by lemma 29, $|\hat{\varsigma}|_{\text{al}} \in succ(|\hat{\varsigma}'|_{\text{al}})$. Thus, in line 7 or 13, $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ was propagated but not put in $Summary$.

By lemma 28, $CE_p(\hat{\varsigma}) \neq \emptyset$. Let $\hat{\varsigma}_2 \in CE_p(\hat{\varsigma})$. By *IH*, $(|\hat{\varsigma}_2|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}})$ was examined. We proceed by cases on whether $(|\hat{\varsigma}_1|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ or $(|\hat{\varsigma}_2|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}})$ was examined first.

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

If $\hat{\varsigma}_2$ has a predecessor $\hat{\varsigma}_3$ which is a tail call, then each $\hat{\varsigma}_4$ in $CE_p^*(\hat{\varsigma}_3)$ is also in $CE_p^*(\hat{\varsigma})$. We must show that $(|\hat{\varsigma}_4|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ 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.) The subpath $\hat{\varsigma}_4 \rightsquigarrow^+ \hat{\varsigma}_3$ has the form $\hat{\varsigma}_4 \rightsquigarrow^+ c_1 \rightsquigarrow e_1 \rightsquigarrow^+ c_2 \rightsquigarrow e_2 \rightsquigarrow^+ \ldots \rightsquigarrow^+ c_n \rightsquigarrow e_n \rightsquigarrow^+ \hat{\varsigma}_3$, where $n \geqslant 0$, c_is are tail calls, e_is are entries and $e_i \in CE_p(c_{i+1})$.
When $n = 0$, $\hat{\varsigma}_4 \in CE_p(\hat{\varsigma}_3)$. When $n > 0$, $e_n \in CE_p(\hat{\varsigma}_3)$ and $\hat{\varsigma}_4 \in CE_p(c_1)$.
We assume that $n > 0$. (For $n = 0$, the proof is simpler.)
First, we show the theorem for $(|e_n|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$; we take cases depending on whether $(|e_n|_{\text{al}}, |\hat{\varsigma}_3|_{\text{al}})$ or $(|\hat{\varsigma}_2|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ was processed first.

- $(|e_n|_{\text{al}}, |\hat{\varsigma}_3|_{\text{al}})$ was first
  By lemma 29, $|\hat{\varsigma}_2|_{\text{al}} \in succ(|\hat{\varsigma}_3|_{\text{al}})$. Thus, in line 37, we put $(|e_n|_{\text{al}}, |\hat{\varsigma}_3|_{\text{al}}, |\hat{\varsigma}_2|_{\text{al}})$ in $TCallers$. When $(|\hat{\varsigma}_2|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ is examined, we follow the `else` branch at line 31. As a result, at line 33 $(|e_n|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ is put in $Summary$ and $Seen$.
- $(|\hat{\varsigma}_2|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ was first
  Then, when $(|e_n|_{\text{al}}, |\hat{\varsigma}_3|_{\text{al}})$ is examined, $(|\hat{\varsigma}_2|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ is in $Summary$. By lemma 29, $|\hat{\varsigma}_2|_{\text{al}} \in succ(|\hat{\varsigma}_3|_{\text{al}})$. Thus, in line 41, $(|e_n|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ 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 $(|e_n|_{\text{al}}, |\hat{\varsigma}|_{\text{al}})$ will be in $Summary$ when it is examined.

By repeating this process n times, we can show that, eventually, for any i the edge $(|e_i|_{\mathrm{al}}, |\hat{\varsigma}|_{\mathrm{al}})$ satisfies the theorem, and the edge $(|\hat{\varsigma}_4|_{\mathrm{al}}, |\hat{\varsigma}|_{\mathrm{al}})$ also satisfies the theorem. □

APPENDIX B

Complexity of the CFA2 workset algorithm

In this appendix, we compute an upper bound of the running time of the algorithm of fig. 4.8.

Let n be the size of the program to be analyzed. We write $|S|$ for the size of a set S.

$|\widetilde{UProc}| = 2^n \;\Rightarrow\; |Heap| = |\widetilde{Frame}| = 2^{n^2}$

$|\widetilde{Eval}| = n \cdot 2^{n^2} \cdot 2^{n^2} = n2^{2n^2}$

$|\widetilde{UApply}| = n \cdot 2^n \cdot 2^{n^2} = n2^{n^2+n}$

$|\widetilde{CApply}| = n \cdot 2^n \cdot 2^{n^2} \cdot 2^{n^2} = n2^{2n^2+n}$

$|Seen| = |\widetilde{UApply}| \cdot (|\widetilde{Eval}| + |\widetilde{UApply}| + |\widetilde{CApply}|) = |\widetilde{UApply}| \cdot O(|\widetilde{CApply}|) = O(n^2 2^{3n^2+2n})$

$|Summary| = |\widetilde{UApply}| \cdot |\widetilde{Eval}| = O(n^2 2^{3n^2+n})$

From rule $[\widetilde{\text{UEA}}]$, we see that a $\widetilde{UEval}$ state can have at most n successors, one for each user lambda in the program. Therefore,

$|Callers| = |\widetilde{UApply}| \cdot |\widetilde{Eval}| \cdot n = n^3 2^{3n^2+2n}$

$|TCallers| = |Callers|$

All final states have $halt$ in operator position and an empty stack. Thus,

$|Final| = 2^n \cdot 2^{n^2} = 2^{n^2+n}$

We can compare two elements of $\widehat{UProc}$ for equality in time $O(n)$, which implies that we can compare two heaps or two frames for equality in time $O(n^2)$. Therefore, we can compare two local states for equality pointwise in time $O(n^2)$. Last, we also need time $O(n^2)$ to compare pairs or triples of states, *e.g.*, elements of $Seen$ or $Callers$. So, to test membership in $Seen$ in the function `Propagate` takes time $O(n^2 \cdot |Seen|) = O(n^4 2^{3n^2+2n})$. The cost of a call to `Update` is also $O(n^4 2^{3n^2+2n})$.

There are four cases for $\tilde{\varsigma}_2$ in the algorithm. We now compute the cost of each case.

- $\tilde{\varsigma}_2$ is an entry, $\widetilde{CApply}$ or inner $\widetilde{CEval}$
  The number elements in $Seen$ that fall into this case is $|\widetilde{UApply}| \cdot (|\widetilde{UApply}| + |\widetilde{CApply}| + |\widetilde{CEval}|) = |\widetilde{UApply}| \cdot O(|\widetilde{CApply}|) = O(n^2 2^{3n^2+2n})$.
  To find the cost of processing one element, note that each $\tilde{\varsigma}_2$ has one successor, so we may search $Seen$ once, in time $O(n^4 2^{3n^2+2n})$. Thus, the total cost for this case is $O(n^2 2^{3n^2+2n}) \cdot O(n^4 2^{3n^2+2n}) = O(n^6 2^{6n^2+4n})$.
- $\tilde{\varsigma}_2$ is a call
  The pairs $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ that fall into this case are $|\widetilde{UApply}| \cdot |\widetilde{Eval}| = O(n^2 2^{3n^2+n})$.
  By rule $[\widetilde{\text{UEA}}]$, each $\tilde{\varsigma}_2$ can have n successors $\tilde{\varsigma}_3$.
  Line 10 involves searching $Seen$, so it costs $O(n^4 2^{3n^2+2n})$.
  Line 11 involves searching $Callers$, so it costs $O(n^5 2^{3n^2+2n})$.
  The body of the loop in line 12 costs $O(n^4 2^{3n^2+2n})$ and since $\tilde{\varsigma}_3$ is fixed and $\tilde{\varsigma}_4$ is in $\widetilde{Eval}$, the loop costs $|\widetilde{Eval}| \cdot O(n^4 2^{3n^2+2n}) = O(n^5 2^{5n^2+2n})$.
  Thus, line 12 dominates the cost of lines 10 – 12 and the loop in line 9 costs $n \cdot O(n^5 2^{5n^2+2n}) = O(n^6 2^{5n^2+2n})$.
  The total cost for this case is $O(n^2 2^{3n^2+n}) \cdot O(n^6 2^{5n^2+2n}) = O(n^8 2^{8n^2+3n})$.
- $\tilde{\varsigma}_2$ is an Exit-Ret
  The possible $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ pairs are $|\widetilde{UApply}| \cdot |\widetilde{Eval}| = O(n^2 2^{3n^2+n})$.
  The cost of lines 14 – 19 is dominated by the loops in lines 18 and 19.
  The heap does not change in the $[\widetilde{\text{UEA}}]$ transition, so each $\tilde{\varsigma}_1$ can have $|\widetilde{Eval}|/2^{n^2} = n2^{n^2}$ predecessors $\tilde{\varsigma}_4$.
  Thus, the loop in line 18 is executed $|\widetilde{Eval}| \cdot O(n2^{n^2}) = O(n^2 2^{3n^2})$ times, so it costs $O(n^2 2^{3n^2}) \cdot O(n^4 2^{3n^2+2n}) = O(n^6 2^{6n^2+2n})$.
  Line 19 costs the same as line 18.
  The total cost for this case is $O(n^2 2^{3n^2+n}) \cdot O(n^6 2^{6n^2+2n}) = O(n^8 2^{9n^2+3n})$.
- $\tilde{\varsigma}_2$ is an Exit-TC
  This case costs the same as when $\tilde{\varsigma}_2$ is a call.

The overall cost of the algorithm is the sum of the costs of the four cases, which is dominated by the Exit-Ret case, so the algorithm costs $O(n^8 2^{9n^2+3n})$.[1]

[1] In a previous article (LMCS 2011), we mistakenly wrote that the body of the loop in line 19 costs $|Seen|$ because we did not account for the time it takes to compare two elements of $Seen$. Nevertheless, $O(n^8 2^{9n^2+3n})$ is a tighter upper bound than the one in the LMCS article for two reasons. First, we multiply the cost of each case for $\tilde{\varsigma}_2$ only with the relevant pairs of states, not with all edges in $Seen$. Second, we use rule $[\widetilde{\text{UEA}}]$ to find a tighter bound for the number of iterations of the loop in line 18.

Bibliography

[1] Ole Agesen. The Cartesian Product Algorithm: simple and precise type inference of parametric polymorphism. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 2–26, 1995.

[2] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas W. Reps, and Mihalis Yannakakis. Analysis of recursive state machines. *Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):786–818, 2005.

[3] Lars Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, 1994.

[4] Andrew W. Appel. *Compiling with continuations*. Cambridge University Press, 1992.

[5] Gogul Balakrishnan and Thomas Reps. Recency-abstraction for heap-allocated storage. In *Static Analysis Symposium (SAS)*, pages 221–239, 2006.

[6] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: application to model-checking. In *International Conference on Concurrency Theory (CONCUR)*, pages 135–150, 1997.

[7] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 243–262, 2009.

[8] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Programming Language Design and Implementation (PLDI)*, pages 363–374, 2009.

[9] Jong-Deok Choi, Michael G. Burke, and Paul R. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Principles of Programming Languages (POPL)*, pages 232–245, 1993.

[10] William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label or a simple optimizing compiler for Scheme. In *LISP and Functional Programming*, pages 128–139, 1994.

[11] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[12] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages (POPL)*, pages 269–282, 1979.

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

[14] Olivier Danvy and Andrzej Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.

[15] Olivier Danvy and Julia L. Lawall. Back to direct style II: first-class continuations. In *LISP and Functional Programming*, pages 299–310, 1992.

[16] Saumya K. Debray and Todd A. Proebsting. Interprocedural control flow analysis of first-order programs with tail-call optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):568–585, 1997.

[17] Christopher Earl, Matthew Might, and David Van Horn. Pushdown control-flow analysis of higher-order programs. In *Workshop on Scheme and Functional Programming*, 2010.

[18] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs.

In *Programming Language Design and Implementation (PLDI)*, pages 85–96, 1998.

[19] Manuel Fähndrich and Jakob Rehof. Type-based flow analysis and context-free language reachability. *Mathematical Structures in Computer Science*, 18(5):823–894, 2008.

[20] Matthias Felleisen. The theory and practice of first-class prompts. In *Principles of Programming Languages (POPL)*, pages 180–190, 1988.

[21] Alain Finkel, Bernard Willems, and Pierre Wolper. A direct symbolic approach to model checking pushdown systems. In *Verification of Infinite State Systems (Infinity)*, pages 27–37, 1997.

[22] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Programming Language Design and Implementation (PLDI)*, pages 237–247, 1993.

[23] Daniel Friedman and Mitchell Wand. *Essentials of Programming Languages, 3rd edition*. MIT Press, 2008.

[24] Salvatore Guarnieri and Benjamin Livshits. GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, pages 151–197, 2009.

[25] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for Ajax intrusion detection. In *International Conference on World Wide Web (WWW)*, pages 561–570, 2009.

[26] Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In *Programs as Data Objects*, pages 63–83, 2001.

[27] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Programming Language Design and Implementation (PLDI)*, pages 97–105, 1998.

[28] Phillip Heidegger and Peter Thiemann. Recency types for dynamically-typed, object-based languages. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2009.

[29] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *Programming Language Design and Implementation (PLDI)*, pages 254–263, 2001.

[30] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of Javascript web applications. In *Foundations of Software Engineering (FSE)*, pages 59–69, 2011.

[31] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Static Analysis Symposium (SAS)*, pages 238–255, 2009.

[32] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Static Analysis Symposium (SAS)*, pages 320–339, 2010.

[33] Matt Kaufmann, Panagiotis Manolios, and J Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[34] Andrew Kennedy. Compiling with continuations, continued. In *International Conference on Functional Programming (ICFP)*, pages 177–190, 2007.

[35] Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. In *Compiler Construction (CC)*, pages 125–140, 1992.

[36] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Principles of Programming Languages (POPL)*, pages 416–428, 2009.

[37] Naoki Kobayashi. Higher-order model checking: From theory to practice. In *Logic in Computer Science (LICS)*, pages 219–224, 2011.

[38] David Kranz. *ORBIT: An optimizing compiler for Scheme*. PhD thesis, Yale University, 1988.

[39] 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.

[40] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1):1–53, 2008.

[41] Mario Méndez-Lojo, Jorge A. Navas, and Manuel V. Hermenegildo. A flexible, (C)LP-based approach to the analysis of object-oriented programs. In *Logic-Based Program Synthesis and Transformation (LOPSTR)*, pages 154–168, 2007.

[42] Matthew Might. *Environment analysis of higher-order languages*. PhD thesis, Georgia Institute of Technology, 2007.

[43] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Programming Language Design and Implementation (PLDI)*, pages 305–315, 2010.

[44] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *International Symposium on Software Testing and Analyses (ISSTA)*, pages 1–11, 2002.

[45] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996.

[46] Kalyan Muthukumar and Manuel Hermenegildo. Determination of variable dependence information through abstract interpretation. In *North American Conference on Logic Programming (NACLP)*, pages 166–185, 1989.

[47] Jorge A. Navas, Mario Méndez-Lojo, and Manuel V. Hermenegildo. A generic, context sensitive analysis framework for object oriented programs. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2007.

[48] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[49] Hanne Riis Nielson and Flemming Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Principles of Programming Languages (POPL)*, pages 332–345, 1997.

[50] Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: from polymorphic subtyping to CFL-reachability. In *Principles of Programming Languages (POPL)*, pages 54–66, 2001.

[51] John Reppy. Type-sensitive control-flow analysis. In *Workshop on ML*, pages 74–83, 2006.

[52] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Principles of Programming Languages (POPL)*, pages 49–61, 1995.

[53] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Programming Language Design and Implementation (PLDI)*, pages 1–12, 2010.

[54] 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 (ICFP)*, pages 317–328, 2009.

[55] Atanas Rountev, Ana Milanova, and Barbara Ryder. Points-to analysis for java using annotated constraints. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 43–55, 2001.

[56] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *LISP and Functional Programming*, pages 288–298, 1992.

[57] Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.

[58] Manuel Serrano. Control flow analysis: a functional languages compilation paradigm. In *Symposium on Applied Computing (SAC)*, pages 118–122, 1995.

[59] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven Muchnick and Neil Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981.

[60] Olin Shivers. Control flow analysis in Scheme. In *Programming Language Design and Implementation (PLDI)*, pages 164–174, 1988.

[61] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991.

[62] Olin Shivers. The semantics of Scheme control-flow analysis. In *Partial Evaluation and Program Manipulation (PEPM)*, pages 190–198, 1991.

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

[64] Olin Shivers. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. In *SIGPLAN Notices, special issue: 20 years of PLDI (1979 - 1999): a selection*. ACM, 2004.

[65] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Principles of Programming Languages (POPL)*, pages 17–30, 2011.

[66] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Programming Language Design and Implementation (PLDI)*, pages 387–400, 2006.

[67] Guy L. Steele. Rabbit: A compiler for Scheme. Master's thesis, Massachusetts Institute of Technology, 1978.

[68] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages (POPL)*, pages 32–41, 1996.

[69] Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symposium on Programming (ESOP)*, pages 408–422, 2005.

[70] David Van Horn and Harry Mairson. Deciding k-CFA is complete for EXPTIME. In *International Conference on Functional Programming (ICFP)*, pages 275–282, 2008.

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

[72] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In *Principles of Programming Languages (POPL)*, pages 435–445, 1994.

[73] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 97–118, 2005.

[74] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Programming Language Design and Implementation (PLDI)*, pages 131–144, 2004.

[75] Andrew Wright and Suresh Jagannathan. Polymorphic Splitting: an effective polyvariant flow analysis. *Transactions on programming languages and systems (TOPLAS)*, 20(1):166–207, 1998.

[76] Jianwen Zhu. Symbolic pointer analysis. In *International Conference on Computer-aided Design (ICCAD)*, pages 150–157, 2002.