Pushdown Control-Flow Analysis for Free

Thomas Gilray Steven Lyde Michael D. Adams Matthew Might David Van Horn

University of Utah, USA University of Maryland, USA

{tgilray,lyde,adamsmd,might}@cs.utah.edu, dvanhorn@cs.umd.edu

Abstract

Traditional control-flow analysis (CFA) for higher-order languages introduces spurious connections between callers and callees, and different invocations of a function may pollute each other's return flows. Recently, three distinct approaches have been published that provide perfect call-stack precision in a computable manner: CFA2, PDCFA, and AAC. Unfortunately, implementing CFA2 and PDCFA requires significant engineering effort. Furthermore, all three are computationally expensive. For a monovariant analysis, CFA2 is in $O(2^n)$, PDCFA is in $O(n^6)$, and AAC is in $O(n^8)$.

In this paper, we describe a new technique that builds on these but is both straightforward to implement and computationally inexpensive. The crucial insight is an unusual state-dependent allocation strategy for the addresses of continuations. Our technique imposes only a constant-factor overhead on the underlying analysis and costs only $O(n^3)$ in the monovariant case. We present the intuitions behind this development, benchmarks demonstrating its efficacy, and a proof of the precision of this analysis.

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

Keywords Static analysis; Control-flow analysis; Abstract interpretation; Pushdown analysis; Store-allocated continuations

1. Introduction

Recent developments in the static analysis of higher-order languages make it possible to obtain perfect precision in modeling the call stack. This allows calls and returns to be matched up precisely and avoids spurious return flows. Consider the following Racket code, which binds an identity function and applies it on two distinct values:

```
(let* ([id (lambda (x) x)]
       [y (id #t)]
       [z (id #f)])
  ...)
```

Without a precise modeling of the call stack, the value `#f` can spuriously flow to the variable `y`, even when a technique like call sensitivity initially keeps them separate.

Copyright © ACM, 2016. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *POPL '16: Proceedings of the 43rd annual ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2016, http://dx.doi.org/10.1145/2837614.2837631.

PoPL '16, January 20–22, 2016, St. Petersburg, FL, USA.
Copyright © 2016 ACM . . . $15.00.
http://dx.doi.org/10.1145/2837614.2837631

To avoid this imprecision, Vardoulakis and Shivers [16] introduces a *context-free approach* (as in context-free languages, not context sensitivity) to program analysis with CFA2. This technique provides a computable, although exponential-time, method for obtaining perfect stack precision for monovariant analyses of continuation-passing-style programs. Two other approaches, PDCFA and AAC, build on this work by enabling polyvariant (e.g., context sensitive) analysis of direct-style programs and do so at only a polynomial-factor increase to the run-time complexity of the underlying analysis.

Earl *et al.* [4] presents a *pushdown control-flow analysis* (PDCFA), which improves on traditional control-flow analysis by annotating edges in the state graph with stack actions (i.e., push and pop) that implicitly represent precise call stacks. But, this method obtains its precision at a substantial increase in worst-case complexity. For example, a monovariant PDCFA is in $O(n^6)$ where its finite-state equivalent is in $O(n^3)$. Unfortunately, PDCFA also requires significant machinery and presents challenges to engineers responsible for constructing and maintaining such analyses.

Johnson and Van Horn [8] presents *abstracting abstract control* (AAC), a refinement of store-allocated continuations with the established finite-state method of merging stack frames into the store, and defines an allocator that is precise enough to avoid all spurious merging. The key advantage of this method is that it is trivial to implement in existing analysis frameworks that use store-allocated continuations and comes at the cost of changing roughly one line of code. Unfortunately, AAC is more computationally complex than PDCFA as even in the monovariant case it is in $O(n^8)$.

We draw on the lessons learned from all three approaches and present a technique for obtaining perfect call-stack precision at only a constant-factor increase to run-time complexity over traditional finite-state analysis (i.e., *for free* in terms of complexity) and requiring no refactoring of analyses already using store-allocated continuations (i.e., *for free* in terms of labor).

1.1 Contributions

We contribute an efficient method for obtaining a perfectly precise modeling of the call stack in static analyses. Specifically:

- We present a novel technique for obtaining perfect call-stack precision at no asymptotic cost to run-time complexity and requiring only a trivial change to analyses already using store-allocated continuations. In the monovariant case, our analysis is in $O(n^3)$, the same complexity class as a traditional 0-CFA.
- We illustrate the intuition behind our approach and explain why previous PTIME methods (PDCFA and AAC) fail to exploit it.
- We describe our implementation and provide benchmarks that demonstrate its efficacy.
- We define a relationship between our technique and a static analysis that uses unbounded stacks and use it to prove the precision of our method.

1.2 Outline

Section 2 defines a simple direct-style language and its operational semantics. It presents the relevant background on abstract interpretation using abstract machines, soundness, store widening, and concepts necessary to understanding our technique. We close this section by giving a walkthrough of the above example, illustrating precisely how values become merged in a traditional analysis.

In section 3, we formalize an incomputable static analysis that defines what is meant by perfect stack precision. This analysis loses no precision in its modeling of the call stack but requires an infinite number of unbounded stacks to be explored. We then review PDCFA and AAC, the existing polynomial-time approaches to obtaining an equivalent stack precision.

In section 4, we formalize our technique, give the intuitions that led us to it, and explain how it relates to each of the analyses described in section 3. We describe our implementation and present both monovariant and call-sensitive allocation benchmark results that compare the complexity and precision of our technique to that of ACC.

Section 5 provides a formal relationship between the unbounded-stack machine of section 3 and our improved finite-state analysis. We use this relationship to prove the perfect precision of our method.

2. Background

Static analysis by abstract interpretation proves properties of a program by running its code through an interpreter powered by an *abstract semantics* that approximates the behavior of a *concrete semantics*. This process is a general method for analyzing programs and serves applications such as program verification, malware/vulnerability detection, and compiler optimization, among others [1–3, 10] The *abstracting abstract machines* (AAM) approach uses abstract interpretation of abstract machines for *control-flow analysis* (CFA) of functional (higher-order) programming languages [9, 12, 15]. The AAM methodology allows a high degree of control over how program states are represented and is easy to instrument.

In this section, we review operational semantics and abstract interpretation using AAM along with other concepts we will require as we progress. We present a concrete interpretation of a simple direct-style language, a traditional finite-state abstraction, and a store-widened polynomial-time analysis. We then explore the return-flow merging problem in greater detail.

2.1 Concrete Semantics

We will be using the direct-style (call-by-value, untyped) λ-calculus in administrative-normal-form (ANF) [6].

$$
\begin{array}{rll}
e \in \mathsf{Exp} ::= & \texttt{(let ([}x\ \texttt{(}f\ \mathit{æ}\texttt{)])}\ e\texttt{)} & \text{[call]} \\
\mid & \mathit{æ} & \text{[return]} \\
f, \mathit{æ} \in \mathsf{AExp} ::= & x \mid lam & \text{[atomic expressions]} \\
lam \in \mathsf{Lam} ::= & \texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e\texttt{)} & \text{[lambda abstractions]} \\
x, y \in \mathsf{Var} & \text{is a set of identifiers} & \text{[variables]}
\end{array}
$$

All intermediate expressions are administratively `let`-bound, and the order of operations is made explicit as a stack of such `let`s. This not only simplifies our semantics, but is convenient for analysis as every intermediate expression can naturally be given a unique identifier. Additional core forms permitting mutation, recursive binding, conditional branching, tail calls, and primitive operations add complexity, but do not complicate the technique we aim to discuss and so are left out.

Our concrete interpreter operates over machine states ς.

$$
\begin{array}{rll}
\varsigma \in \Sigma \triangleq & \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont} & \text{[states]} \\
\rho \in \mathit{Env} \triangleq & \mathsf{Var} \rightharpoonup \mathit{Addr} & \text{[environments]} \\
\sigma \in \mathit{Store} \triangleq & \mathit{Addr} \rightharpoonup \mathit{Clo} & \text{[stores]} \\
\mathit{clo} \in \mathit{Clo} \triangleq & \mathsf{Lam} \times \mathit{Env} & \text{[closures]} \\
\kappa \in \mathit{Kont} \triangleq & \mathit{Frame}^* & \text{[stacks]} \\
\phi \in \mathit{Frame} \triangleq & \mathsf{Var} \times \mathsf{Exp} \times \mathit{Env} & \text{[stack frames]} \\
a \in \mathit{Addr} & \text{is an infinite set} & \text{[addresses]}
\end{array}
$$

Binding environments (ρ) map variables in scope to a representative address (a). Value stores (σ) map these addresses to a program value. (For pure λ-calculus, all values are closures.) Both are partial functions that are incrementally extended with new points. A closure (clo) pairs a syntactic lambda with an environment over which it is closed. Continuations (κ) are unbounded sequences of stack frames. Each stack frame (ϕ) contains a variable to bind, an expression control returns to, and an environment to reinstate. Addresses (a) may be drawn from any set which permits us to generate an arbitrary number of fresh values (e.g., $\mathbb{N}$).

We define a helper $\mathcal{A} : \mathsf{AExp} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Clo}$ for atomic-expression evaluation:

$$
\begin{array}{rll}
\mathcal{A}(x, \rho, \sigma) \triangleq & \sigma(\rho(x)) & \text{[variable lookup]} \\
\mathcal{A}(lam, \rho, \sigma) \triangleq & (lam, \rho) & \text{[closure creation]}
\end{array}
$$

A concrete transition relation $(\rightsquigarrow_\Sigma) : \Sigma \rightharpoonup \Sigma$ defines the operation of this machine by determining at most one successor for any given predecessor state. The machine stops when the end of a program's execution is reached or when given an invalid state. Call sites transition according to the following transition rule:

$$
\begin{array}{c}
(\texttt{(let ([}y\ \texttt{(}f\ \mathit{æ}\texttt{)])}\ e\texttt{)}, \rho, \sigma, \kappa) \rightsquigarrow_\Sigma (e', \rho', \sigma', \phi{:}\kappa), \text{where} \\
\phi = (y, e, \rho) \\
(\texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e'\texttt{)}, \rho_\lambda) = \mathcal{A}(f, \rho, \sigma) \\
\rho' = \rho_\lambda[x \mapsto a] \\
\sigma' = \sigma[a \mapsto \mathcal{A}(\mathit{æ}, \rho, \sigma)] \\
a \text{ is a } \textit{fresh} \text{ address}
\end{array}
$$

A new frame ϕ is pushed onto the stack for eventually returning to the body of this `let`-form. The atomic expression f is either a lambda-form or a variable-reference and is evaluated to a closure by our helper $\mathcal{A}$. In our notation, ticks are used to uniquely name identifiers that may be different. These do not have any bearing on the variable's domain, but where possible will hint at usage (e.g., a single tick for a successor's components). A subscript may be more significant, but we will be careful to point it out. This is not the case for ρ_λ, which is used to name whatever environment was drawn from the closure for f. This is simply an environment distinct from ρ and ρ'. We generate a *fresh* address a (any address such that $a \notin \mathit{dom}(\sigma)$) and update ρ_λ with a mapping $x \mapsto a$ to produce the successor environment ρ'. Likewise, the prior store σ is extended at this address with the value for $\mathit{æ}$ to produce σ'.

Return points transition according to a second rule:

$$
\begin{array}{c}
(\mathit{æ}, \rho, \sigma, \phi{:}\kappa) \rightsquigarrow_\Sigma (e, \rho', \sigma', \kappa), \text{where} \\
\phi = (x, e, \rho_\kappa) \\
\rho' = \rho_\kappa[x \mapsto a] \\
\sigma' = \sigma[a \mapsto \mathcal{A}(\mathit{æ}, \rho, \sigma)] \\
a \text{ is a } \textit{fresh} \text{ address}
\end{array}
$$

The top stack frame ϕ is decomposed and its environment ρ_κ extended with a fresh address a to produce ρ'. Likewise, the store

is extended at this address with the value for æ to produce σ'. The expression e in the top stack frame is reinstated at ρ', and σ' is put atop the predecessor's stack tail κ.

To fully evaluate a program e_0 using these transition rules, we *inject* it into our state-space using a helper $\mathcal{I} : \mathsf{Exp} \to \Sigma$:

$$\mathcal{I}(e) \triangleq (e, \varnothing, \varnothing, \epsilon)$$

We perform the standard lifting of $(\rightsquigarrow_\Sigma)$ to obtain a collecting semantics defined over sets of states:

$$s \in S \triangleq \mathcal{P}(\Sigma)$$

Our collecting relation $(\rightsquigarrow_s)$ is a monotonic, total function that gives a set including the trivially reachable state $\mathcal{I}(e_0)$ plus the set of all states immediately succeeding those in its input.

$$s \rightsquigarrow_s s' \triangleq s' = \{\varsigma' \mid \varsigma \in s \wedge \varsigma \rightsquigarrow_\Sigma \varsigma'\} \cup \{\mathcal{I}(e_0)\}$$

If the program e_0 terminates, iteration of $(\rightsquigarrow_s)$ from $\bot$ (i.e., the empty set $\varnothing$) does as well. That is, $(\rightsquigarrow_s)^n(\bot)$ is a fixed point containing e_0's full program trace for some $n \in \mathbb{N}$ whenever e_0 is a terminating program. No such n is guaranteed to exist in the general case (when e_0 is a non-terminating program) as our language (the untyped λ-calculus) is Turing-complete, our semantics is fully precise, and the state-space we defined is infinite.

2.2 Abstract Semantics

We are now ready to design a computable approximation of the exact program trace using an abstract semantics. Previous work has explored a wide variety of approaches to systematically abstracting a semantics like these [9, 12, 15]. Broadly construed, the nature of these changes is to simultaneously finitize the domains of our machine while introducing non-determinism both into the transition relation (multiple successor states may immediately follow a predecessor state) and the store (multiple values may be indicated by a single address). We use a finite state space to ensure computability. However, to justify that a semantics defined over this finite machine is soundly approximating our concrete semantics (for a defined notion of abstraction), we must also modify our finite states so that a potentially infinite number of concrete states may abstract to a single finite state. We will use this term *finite state* to differentiate from other kinds of machine states. Components unique to this finite-state machine wear tildes:

$$\begin{aligned}
\tilde{\varsigma} \in \tilde{\Sigma} &\triangleq \mathsf{Exp} \times \widetilde{Env} \times \widetilde{Store} && \text{[states]}\\
&\quad \times \widetilde{KStore} \times \widetilde{Addr} &&\\
\tilde{\rho} \in \widetilde{Env} &\triangleq \mathsf{Var} \rightharpoonup \widetilde{Addr} && \text{[environments]}\\
\tilde{\sigma} \in \widetilde{Store} &\triangleq \widetilde{Addr} \to \tilde{D} && \text{[stores]}\\
\tilde{d} \in \tilde{D} &\triangleq \mathcal{P}(\widetilde{Clo}) && \text{[flow-sets]}\\
\widetilde{clo} \in \widetilde{Clo} &\triangleq \mathsf{Lam} \times \widetilde{Env} && \text{[closures]}\\
\tilde{\sigma}_\kappa \in \widetilde{KStore} &\triangleq \widetilde{Addr} \to \tilde{K} && \text{[continuation stores]}\\
\tilde{k} \in \tilde{K} &\triangleq \mathcal{P}(\widetilde{Kont}) && \text{[kont-sets]}\\
\tilde{\kappa} \in \widetilde{Kont} &\triangleq \widetilde{Frame} \times \widetilde{Addr} && \text{[continuations]}\\
\tilde{\phi} \in \widetilde{Frame} &\triangleq \mathsf{Var} \times \mathsf{Exp} \times \widetilde{Env} && \text{[stack frame]}\\
\tilde{a}, \tilde{a}_\kappa \in \widetilde{Addr} &\text{ is a finite set} && \text{[addresses]}
\end{aligned}$$

There were two fundamental sources of unboundedness in the concrete machine: the value store (with an infinite domain of addresses), and the current continuation (modeled as an unbounded list of stack frames). We bound the value store ($\tilde{\sigma}$) by restricting its domain to a finite set of addresses ($\tilde{a}$), but we permit a *set* of abstract closures ($\widetilde{clo}$) at each. We finitize the stack similarly by threading it through the store as a linked list. A continuation is thus represented by an address. This address points to a *set* of topmost frames, each paired with the address of its continuation in turn (i.e., that stack's tail). We separate the continuation store ($\tilde{\sigma}_\kappa$) from the value store ($\tilde{\sigma}$) to maintain simplicity as we progress.

Abstract environments ($\tilde{\rho}$) change only because our address set is now finite. Abstract closures ($\widetilde{clo}$) are approximate only by virtue of their environments using these abstract addresses. For each such $\tilde{a}$, the finite value store ($\tilde{\sigma}$) denotes a *flow set* ($\tilde{d}$) of closures. At each point, a continuation store ($\tilde{\sigma}_\kappa$) has a set of continuations ($\tilde{k}$). Like closures, each abstract frame ($\tilde{\phi}$) is approximate only by virtue of its abstracted environment. An abstract continuation ($\tilde{\kappa}$) pairs a frame with an address ($\tilde{a}_\kappa$) for the stack underneath.

As before, we define a helper for abstract atomic evaluation, $\tilde{\mathcal{A}}$:

$$\tilde{\mathcal{A}} : \mathsf{AExp} \times \widetilde{Env} \times \widetilde{Store} \rightharpoonup \tilde{D}$$

$$\begin{aligned}
\tilde{\mathcal{A}}(x, \tilde{\rho}, \tilde{\sigma}) &\triangleq \tilde{\sigma}(\tilde{\rho}(x)) && \text{[variable lookup]}\\
\tilde{\mathcal{A}}(lam, \tilde{\rho}, \tilde{\sigma}) &\triangleq \{(lam, \tilde{\rho})\} && \text{[closure creation]}
\end{aligned}$$

Note that atomic evaluation of a lambda expression new yields a set containing a single element for the closure of that lambda.

Because our address domain is now finite, multiple concrete allocations need to be represented by a single abstract address. There are a variety of sound strategies for doing this. Each strategy corresponds to a distinct style of analysis and is amenable to easy implementation by defining an auxiliary $\widetilde{alloc}$ helper to encapsulate these differences in behavior. Given the variable for which to allocate and the finite state performing the allocation, the abstract allocator returns an address:

$$\widetilde{alloc} : \mathsf{Var} \times \tilde{\Sigma} \to \widetilde{Addr}$$

One such behavior is to simply return the variable itself (as a 0-CFA would):

$$\widetilde{alloc}_0(x, \tilde{\varsigma}) \triangleq x$$

Using $\widetilde{alloc}_0$ would tune our finite-state semantics to the *monovariant* analysis style (also called zeroth-order CFA), a form of context-insensitive analysis. In a monovariant analysis, every closure that is bound to a variable x at any point during a concrete execution ends up being represented in a single flow set when the analysis is complete.

Because we are also now store-allocating continuations and distinguishing a top-level continuation store, we likewise distinguish an abstract allocator specifically for addresses in this store:

$$\widetilde{alloc}_\kappa : \tilde{\Sigma} \times \mathsf{Exp} \times \widetilde{Env} \times \widetilde{Store} \to \widetilde{Addr}$$

A standard choice is to allocate based on the target expression:

$$\widetilde{alloc}_{\kappa 0}((e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa), e', \tilde{\rho}', \tilde{\sigma}') \triangleq e'$$

We provide to this function all the information known about the transition being made. The value-store allocator is invoked before a successor $\tilde{\rho}'$ or $\tilde{\sigma}'$ is constructed. However, when calling the continuation-store allocator, we provide information about the target state being transitioned to. The choice of e' for allocating a continuation address makes sense considering the entry point of a function should know where it is returning. In fact, when performing an analysis of a continuation-passing-style (CPS) language, e' also would naturally be the choice inherited from a monovariant value-store allocator (assuming an alpha-renaming such that every x is unique to a single binding point).

We may now define a non-deterministic finite-state transition relation $(\rightsquigarrow_{\tilde{\Sigma}}) \subseteq \tilde{\Sigma} \times \tilde{\Sigma}$. Call sites transition as follows.

$$\overbrace{((\texttt{let ([}y\ (f\ \texttt{æ})\texttt{])}\ e), \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa)}^{\tilde{\varsigma}} \rightsquigarrow_{\tilde{\Sigma}} (e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}'_\kappa), \text{ where}$$

$$\begin{aligned}
((\lambda\ (x)\ e'), \tilde{\rho}_\lambda) &\in \tilde{\mathcal{A}}(f, \tilde{\rho}, \tilde{\sigma}) \\
\tilde{\rho}' &= \tilde{\rho}_\lambda[x \mapsto \tilde{a}] \\
\tilde{\sigma}' &= \tilde{\sigma} \sqcup [\tilde{a} \mapsto \tilde{\mathcal{A}}(\text{æ}, \tilde{\rho}, \tilde{\sigma})] \\
\tilde{a} &= \widetilde{alloc}(x, \tilde{\varsigma}) \\
\tilde{\sigma}'_\kappa &= \tilde{\sigma}_\kappa \sqcup [\tilde{a}'_\kappa \mapsto \{((y, e, \tilde{\rho}), \tilde{a}_\kappa)\}] \\
\tilde{a}'_\kappa &= \widetilde{alloc}_\kappa(\tilde{\varsigma}, e', \tilde{\rho}', \tilde{\sigma}')
\end{aligned}$$

As $\tilde{\mathcal{A}}$ yields a set of abstract closures for f, a successor state is produced for each. Likewise, so each point in the store accumulates all closures bound at that abstract address $\tilde{a}$ and so we faithfully over-approximate all the addresses a that $\tilde{a}$ simulates, we use a join operation when extending the store. The join of two stores distributes point-wise as follows.

$$\tilde{\sigma} \sqcup \tilde{\sigma}' \triangleq \lambda \tilde{a}.\ \tilde{\sigma}(\tilde{a}) \cup \tilde{\sigma}'(\tilde{a})$$
$$\tilde{\sigma}_\kappa \sqcup \tilde{\sigma}'_\kappa \triangleq \lambda \tilde{a}_\kappa.\ \tilde{\sigma}_\kappa(\tilde{a}_\kappa) \cup \tilde{\sigma}'_\kappa(\tilde{a}_\kappa)$$

Instead of generating a fresh address for $\tilde{a}$, we use our abstract allocation policy to select one. To instantiate a monovariant analysis like 0-CFA, this address is simply the syntactic variable x. Likewise, we generate an address for our continuation (a new stack frame atop the current continuation) and extend the continuation store.

The return transition is modified in the same way:

$$\overbrace{(\text{æ}, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa)}^{\tilde{\varsigma}} \rightsquigarrow_{\tilde{\Sigma}} (e, \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}_\kappa, \tilde{a}'_\kappa), \text{ where}$$

$$\begin{aligned}
((x, e, \tilde{\rho}_\kappa), \tilde{a}'_\kappa) &\in \tilde{\sigma}_\kappa(\tilde{a}_\kappa) \\
\tilde{\rho}' &= \tilde{\rho}_\kappa[x \mapsto \tilde{a}] \\
\tilde{\sigma}' &= \tilde{\sigma} \sqcup [\tilde{a} \mapsto \tilde{\mathcal{A}}(\text{æ}, \tilde{\rho}, \tilde{\sigma})] \\
\tilde{a} &= \widetilde{alloc}(x, \tilde{\varsigma})
\end{aligned}$$

Where multiple topmost stack frames are pointed to by $\tilde{a}_\kappa$, this transition yields multiple successors. An updated environment and store are produced as before, but the continuation store remains as it was. The current continuation $\tilde{a}'_\kappa$ reinstated in each successor is the address associated with each topmost stack frame.

To approximately evaluate a program according to these abstract semantics, we first define an abstract injection function, $\tilde{\mathcal{I}}$, where the stores begin as functions, $\bot$, that map every abstract address to the empty set.

$$\tilde{\mathcal{I}} : \mathsf{Exp} \to \tilde{\Sigma}$$

$$\tilde{\mathcal{I}}(e) \triangleq (e, \varnothing, \bot, \bot, \tilde{a}_{\text{halt}})$$

The address $\tilde{a}_{\text{halt}}$ can be any otherwise unused address that is never returned by the allocation function. Our machine will eventually be unable to transition into this continuation and will then produce no successors, which simulates the behavior of our concrete machine upon reaching an empty stack (ϵ).

We again lift $(\rightsquigarrow_{\tilde{\Sigma}})$ to obtain a collecting semantics $(\rightsquigarrow_{\tilde{S}})$ defined over sets of states:

$$\tilde{s} \in \tilde{S} \triangleq \mathcal{P}(\tilde{\Sigma})$$

$$\tilde{s} \rightsquigarrow_{\tilde{S}} \tilde{s}' \ \triangleq\ \tilde{s}' = \{\tilde{\varsigma}' \mid \tilde{\varsigma} \in \tilde{s} \wedge \tilde{\varsigma} \rightsquigarrow_{\tilde{\Sigma}} \tilde{\varsigma}'\} \cup \{\tilde{\mathcal{I}}(e_0)\}$$

Our collecting relation $(\rightsquigarrow_{\tilde{S}})$ is a monotonic, total function that gives a set including the trivially reachable finite-state $\tilde{\mathcal{I}}(e_0)$ plus the set of all states immediately succeeding those in its input.

Because $\tilde{\Sigma}$ is now finite, we know the approximate evaluation of even a non-terminating e_0 will terminate. That is, for some $n \in \mathbb{N}$, the value $(\rightsquigarrow_{\tilde{S}})^n(\bot)$ is guaranteed to be a fixed point containing an approximation of e_0's full program trace [14].

2.3 Soundness

An analysis is *sound* if the information it provides about a program represents an accurate bound on the behavior of all possible concrete executions. The kind of control-flow information the finite-state analysis in section 2.2 obtains is a conservative over-approximation of program behavior. It places an upper bound on the propagation of closures though a program.

To establish such a relationship between a concrete and abstract semantics, we use Galois connections. A Galois connection is a pair of functions for abstraction and concretization such that the following holds.

$$\alpha : S \to \tilde{S} \qquad \gamma : \tilde{S} \to S$$

$$\alpha(s) \subseteq \tilde{s} \iff s \subseteq \gamma(\tilde{s})$$

Using this defined notion of simulation, we may show that our abstract semantics approximates the concrete semantics by proving that simulation is preserved across transition:

$$\alpha(s) \subseteq \tilde{s} \wedge s \rightsquigarrow_S s' \implies \tilde{s} \rightsquigarrow_{\tilde{S}} \tilde{s}' \wedge \alpha(s') \subseteq \tilde{s}'$$

Diagrammatically this is:

$$\begin{array}{ccc}
s & \xrightarrow{\rightsquigarrow_S} & s' \\
\subseteq \big\downarrow \alpha & & \subseteq \big\downarrow \alpha \\
\tilde{s} & \xrightarrow{\rightsquigarrow_{\tilde{S}}} & \tilde{s}'
\end{array}$$

Both constructing analyses using Galois connections and proving them sound using Galois connections has been extensively explored in the literature [12, 15]. The analysis style we constructed in section 2.2 has been previously proven sound using the above method [11].

2.4 Store Widening

Various forms of widening and further approximations may be layered on top of this naïve analysis. One such approximation is store widening, which is necessary for our analysis to be tractable (i.e., polynomial time). To see why store widening is necessary, let us consider the complexity of an analysis using $(\rightsquigarrow_{\tilde{S}})$. The height of the power-set lattice $(\tilde{S}, \cup, \cap)$ is the number of elements in $\tilde{\Sigma}$ which is the product of expressions, environments, stores, and addresses. A standard worklist algorithm at most does work proportional to the number of states it can discover [13]. For the imprecise allocators we have defined, analysis run-time is thus in:

$$O(\overbrace{n}^{|\mathsf{Exp}|} \times \overbrace{n}^{|\widetilde{Env}|} \times \overbrace{2^{n^2}}^{|\widetilde{Store}|} \times \overbrace{2^{n^2}}^{|\widetilde{KStore}|} \times \overbrace{n}^{|\widetilde{Addr}|})$$

The number of syntactic points in an input program is in $O(n)$. In the monovariant case, environments map variables to themselves and are isomorphic to the sets of free variables that may be determined for each syntactic point. The number of addresses produced by our monovariant allocators is in $O(n)$ as these are either syntactic variables or expressions. The number of value stores may be visualized as a table of possible mappings from every address to every abstract closure—each may be included in a given store

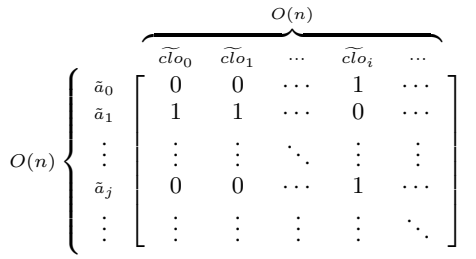


Figure 1. The value space of stores.

or not as seen in figure 1. The number of abstract closures is in $O(n)$ because lambdas uniquely determine a monovariant environment. This times the number of addresses gives $O(n^2)$ possible additions to the value store. The number of continuations is likewise in $O(n)$ because `let`-forms uniquely determine their binding variable, body, and monovariant environment. This times the number of possible addresses gives $O(n^2)$ possible additions to the continuation store.

The crux of the issue is that, in exploring a naïve state-space (where each state is specific to a whole store), we may explore both sides of every diamond in the store lattices. All combinations of possible bindings in a store may need to be explored, including every alternate path up the store lattice. For example, along one explored path we might extend an address $\tilde{a}_1$ with $\widetilde{clo}_1$ before extending it with $\widetilde{clo}_2$, and along another path we might add these closures in the reverse order (i.e., $\widetilde{clo}_2$ before $\widetilde{clo}_1$). We might also extend another address $\tilde{a}_2$ with $\widetilde{clo}_1$ either before or after either of these cases, and so forth. This potential for exponential blow-up is unavoidable without further widening or coarser structural abstraction.

Global-store widening is an essential technique for combating exponential blow up. This lifts the store alongside a set of reachable states instead of nesting them inside states in $\tilde{\Sigma}$. To formalize this, we define new *widened* state spaces that pair a set of reachable *configurations* (states *sans* stores) with a global value store and global continuation store. Instead of accumulating whole stores, and thereby all possible sequences of additions within such stores, the analysis strictly accumulates new values in the store in the same way ($\rightsquigarrow_{\tilde{s}}^{\sim}$) accumulates reachable states in an $\tilde{s}$:

$$\tilde{\xi} \in \tilde{\Xi} \triangleq \tilde{R} \times \widetilde{Store} \times \widetilde{KStore} \quad \text{[state-spaces]}$$
$$\tilde{r} \in \tilde{R} \triangleq \mathcal{P}(\tilde{C}) \quad \text{[reachable configurations]}$$
$$\tilde{c} \in \tilde{C} \triangleq \mathsf{Exp} \times \widetilde{Env} \times \widetilde{Addr} \quad \text{[configurations]}$$

A widened transfer function ($\rightsquigarrow_{\tilde{\Xi}}^{\sim}$) may then be defined that, like ($\rightsquigarrow_{\tilde{s}}^{\sim}$), is a monotonic, total function we may iterate to a fixed point.

$$(\rightsquigarrow_{\tilde{\Xi}}^{\sim}) : \tilde{\Xi} \to \tilde{\Xi}$$

This may be defined in terms of ($\rightsquigarrow_{\tilde{\Sigma}}^{\sim}$), as was ($\rightsquigarrow_{\tilde{s}}^{\sim}$), by transitioning each reachable configuration using the global store to yield a new set of reachable configurations and a set of stores whose least upper bound is the new global store:

$$(\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa) \rightsquigarrow_{\tilde{\Xi}}^{\sim} (\tilde{r}', \tilde{\sigma}', \tilde{\sigma}'_\kappa), \text{ where}$$

$$\tilde{s} = \{\tilde{\varsigma} \mid (e, \tilde{\rho}, \tilde{a}_\kappa) \in \tilde{r} \wedge (e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa) \rightsquigarrow_{\Sigma}^{\sim} \tilde{\varsigma}\} \cup \{\tilde{\mathcal{I}}(e_0)\}$$
$$\tilde{r}' = \{(e, \tilde{\rho}, \tilde{a}_\kappa) \mid (e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa) \in \tilde{s}\}$$
$$\tilde{\sigma}' = \bigsqcup_{(\_,\_,\tilde{\sigma}'',\_,\_)\in\tilde{s}} \tilde{\sigma}'' \qquad \tilde{\sigma}'_\kappa = \bigsqcup_{(\_,\_,\_,\tilde{\sigma}''_\kappa,\_)\in\tilde{s}} \tilde{\sigma}''_\kappa$$

In this definition, an underscore (wildcard) matches anything. The height of the $\tilde{R}$ lattice is linear (as environments are monovariant) and the height of the store lattices are quadratic (as each global store is strictly extended). Each extension of the store may require $O(n)$ transitions because at any given store, we must transition every configuration to be sure to obtain any changes to the store or otherwise reach a fixed point. A traditional worklist algorithm for computing a fixed point is thus cubic:

$$O(\overbrace{n}^{|\tilde{C}|} \times (\overbrace{n^2}^{|\widetilde{Store}|} + \overbrace{n^2}^{|\widetilde{KStore}|}))$$

2.5 Stack Imprecision

To illustrate the effect of an imprecise stack on data-flow and control-flow precision, we first define a more precise 1-call-sensitive (first-order, 1-CFA) allocator. A k-call-sensitive analysis style differentiates bindings to a variable so they are unique to a history of the last k call sites reached before the binding. A history of length $k = 1$ then allocates an address unique to the call site immediately preceeding the binding by using the following allocator.

$$\widetilde{alloc}_1(x, (e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{\kappa})) \triangleq (x, e)$$

Now, using $\widetilde{alloc}_1$, consider the following snippet of code where the variable `id` is already bound to `(λ (x) ⁰x)`:

```
... ¹(let ([y (id #t)])
      ²(let ([z (id #f)])
         ³...))
```

We number these expressions for ease of reference. For example, e_2 refers to the `let`-form that binds `z`, and e_0 to the return point of `id`. We assume the starting configuration for this example is $(e_1, \tilde{\rho}, \tilde{a}_\kappa)$ where $\tilde{\rho}$ and $\tilde{a}_\kappa$ are the binding environment and continuation address at the start of this code. We likewise let $\tilde{\rho}_\lambda$ be the environment of `id`'s closure.

The first call to `id` transitions to evaluate e_0 with the continuation address e_0. This transition reaches the configuration $(e_0, \tilde{\rho}_\lambda[\mathtt{x} \mapsto (\mathtt{x}, e_1)], e_0)$ and binds $(\mathtt{x}, e_1)$ to `#t` and the continuation address e_0 to the continuation $((\mathtt{y}, e_2, \tilde{\rho}), \tilde{a}_\kappa)$, which gives us the following stores:

$$\begin{aligned}\tilde{\sigma} &= \{(\mathtt{x}, e_1) \mapsto \{\mathtt{\#t}\}\} \\ \tilde{\sigma}_\kappa &= \{e_0 \mapsto \{((\mathtt{y}, e_2, \tilde{\rho}), \tilde{a}_\kappa)\}\}\end{aligned}$$

Next, `id` returns and transitions from e_0 to e_2, extending the continuation's environment to $\tilde{\rho}[\mathtt{y} \mapsto (\mathtt{y}, e_0)]$ and reinstating the continuation address $\tilde{a}_\kappa$. This yields a configuration $(e_2, \tilde{\rho}[\mathtt{y} \mapsto (\mathtt{y}, e_0)], \tilde{a}_\kappa)$. This transition binds $(\mathtt{y}, e_0)$ to `#t`, giving us the following stores:

$$\begin{aligned}\tilde{\sigma} &= \{(\mathtt{x}, e_1) \mapsto \{\mathtt{\#t}\}, \\ &\qquad (\mathtt{y}, e_0) \mapsto \{\mathtt{\#t}\}\} \\ \tilde{\sigma}_\kappa &= \{e_0 \mapsto \{((\mathtt{y}, e_2, \tilde{\rho}), \tilde{a}_\kappa)\}\}\end{aligned}$$

Then the second call to `id` transitions to evaluate e_0 with the continuation address e_0 once again (recall the definition of $\widetilde{alloc}_{\kappa 0}$). This transition reaches the configuration $(e_0, \tilde{\rho}_\lambda[\mathtt{x} \mapsto (\mathtt{x}, e_2)], e_0)$, binding $(\mathtt{x}, e_2)$ to `#f` and the continuation address e_0 to the continuation $((\mathtt{z}, e_3, \tilde{\rho}[\mathtt{y} \mapsto (\mathtt{y}, e_0)]), \tilde{a}_\kappa)$, giving us the following stores:

$$\begin{aligned}\tilde{\sigma} &= \{(\mathtt{x}, e_1) \mapsto \{\mathtt{\#t}\}, \\ &\qquad (\mathtt{y}, e_0) \mapsto \{\mathtt{\#t}\}, \\ &\qquad (\mathtt{x}, e_2) \mapsto \{\mathtt{\#f}\}\} \\ \tilde{\sigma}_\kappa &= \{e_0 \mapsto \{((\mathtt{y}, e_2, \tilde{\rho}), \tilde{a}_\kappa), \\ &\qquad\qquad ((\mathtt{z}, e_3, \tilde{\rho}[\mathtt{y} \mapsto (\mathtt{y}, e_0)]), \tilde{a}_\kappa)\}\}\end{aligned}$$

Next, `id` returns and transitions from e_0 to e_3, reinstating the continuation address $\tilde{a}_\kappa$ and extending the continuation's environment to $\tilde{\rho}[\mathtt{y} \mapsto (\mathtt{y}, e_0)][\mathtt{z} \mapsto (\mathtt{z}, e_0)]$. Because e_0 is bound to two continuations, this transition binds $(\mathtt{z}, e_0)$ to `#f` while another spuriously binds $(\mathtt{y}, e_0)$ to `#f`, causing return-flow imprecision in the following stores:

$$\begin{aligned}\tilde{\sigma} \;= \{&(\mathtt{x}, e_1) \mapsto \{\mathtt{\#t}\},\\ &(\mathtt{x}, e_2) \mapsto \{\mathtt{\#f}\},\\ &(\mathtt{y}, e_0) \mapsto \{\mathtt{\#t}, \mathtt{\#f}\},\\ &(\mathtt{z}, e_0) \mapsto \{\mathtt{\#f}\}\}\end{aligned}$$

$$\begin{aligned}\tilde{\sigma}_\kappa = \{e_0 \quad &\mapsto \{((\mathtt{y}, e_2, \tilde{\rho}), \tilde{a}_\kappa)\,,\\ &\qquad ((\mathtt{z}, e_3, \tilde{\rho}\,[\mathtt{y} \mapsto (\mathtt{y}, e_0)]), \tilde{a}_\kappa)\}\}\end{aligned}$$

The address $(\mathtt{y}, e_0)$, representing `y` within e_3, maps to both `#t` and `#f`, even though no concrete execution binds `y` to `#f`. A similar pair of transitions from $(e_0, \tilde{\rho}_\lambda[\mathtt{x} \mapsto (\mathtt{x}, e_1)], e_0)$ (the second of which is prompted by a change in the global continuation store at the address e_0) cause the same conflation for `z`.

Clearly, one solution might be to increase the context sensitivity of our continuation allocator. Consider a continuation allocator $\widehat{alloc}_{\kappa 1}$ that like $\widehat{alloc}_1$ uses a single call site of context and allocates a continuation address (e', e) formed from both the expression being transitioned to, e', and the expresson being transitioned from, e. This results in no spurious merging at return points because continuations are kept as distinct as the 1-call-sensitive value-store addresses we allocate; for other examples, however, even this would not be precise enough. In general, mimicing the style of polyvariance used by the value allocator, will not lead to a perfectly precise continuation allocator.

It seems reasonable from here to suspect that perfect stack precision could always be obtained through a sufficiently precise strategy for polyvariant continuation allocation. The difficulty is in knowing how to obtain this in the general case given an arbitrary value-store allocation strategy. Given that CFA2 and PDCFA promise a fixed method for implementing perfect stack precision, albeit at significant engineering and run-time costs, can perfect stack precision be implemented as a *fixed*, adaptive continuation allocator? In this paper, we both answer this question in the affirmative and show that this leads us not only to a trivial implementation but to only a constant-factor increase in run-time complexity.

3. Perfect Stack Precision

We next formalize what is meant by a static analysis with *perfect stack precision* by using an abstract abstract machine (AAM) [15] with unbounded stacks within each machine configuration. We then review the existing polynomial-time methods for computing an analysis with equivalent precision to this machine: PDCFA and AAC.

3.1 Unbounded-Stack Analysis

In the same manner as previous work on this topic, we formalize perfect stack precision using a static analysis that leaves the structure of stacks fully unabstracted. Each frame of this unbounded stack is itself abstract because its environment is abstract and references the abstracted value store. States and configurations, however, directly contain lists of such frames that are unbounded in length. Environments, closures, stack frames, flow sets, and value stores are otherwise abstracted in the same manner as the finite machine of section 2.2. To differentiate this from the machines we have seen so far, we call this an unbounded-stack machine. Components unique to this machine wear hats:

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &\triangleq \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \times \widehat{Kont} && \text{[states]}\\
\hat{\rho} \in \widehat{Env} &\triangleq \mathsf{Var} \rightharpoonup \widehat{Addr} && \text{[environments]}\\
\hat{\sigma} \in \widehat{Store} &\triangleq \widehat{Addr} \to \hat{D} && \text{[stores]}\\
\hat{d} \in \hat{D} &\triangleq \mathcal{P}(\widehat{Clo}) && \text{[flow-sets]}\\
\widehat{clo} \in \widehat{Clo} &\triangleq \mathsf{Lam} \times \widehat{Env} && \text{[closures]}\\
\hat{\kappa} \in \widehat{Kont} &\triangleq \widehat{Frame}^* && \text{[whole stacks]}\\
\hat{\phi} \in \widehat{Frame} &\triangleq \mathsf{Var} \times \mathsf{Exp} \times \widehat{Env} && \text{[stack frames]}\\
\hat{a} \in \widehat{Addr} &\text{ is a finite set} && \text{[addresses]}
\end{aligned}$$

Our atomic-expression evaluator works just as before:

$$\begin{aligned}
&\hat{\mathcal{A}} : \mathsf{AExp} \times \widehat{Env} \times \widehat{Store} \rightharpoonup \hat{D}\\
&\hat{\mathcal{A}}(x, \hat{\rho}, \hat{\sigma}) \triangleq \hat{\sigma}(\hat{\rho}(x)) && \text{[variable lookup]}\\
&\hat{\mathcal{A}}(lam, \hat{\rho}, \hat{\sigma}) \triangleq \{(lam, \hat{\rho})\} && \text{[closure creation]}
\end{aligned}$$

As does a monovariant allocator:

$$\begin{aligned}
&\widehat{alloc} : \mathsf{Var} \times \hat{\Sigma} \to \widehat{Addr}\\
&\widehat{alloc}_0(x, \hat{\varsigma}) \triangleq x
\end{aligned}$$

This may be tuned to any other allocation strategy as easily as before.

We now define a non-deterministic unbounded-stack-machine transition relation $(\rightsquigarrow_{\hat{\Sigma}}) \subseteq \hat{\Sigma} \times \hat{\Sigma}$ and a rule for call-site transitions:

$$\overbrace{((\mathtt{let}\ ([y\ (f\ \text{æ})])\ e), \hat{\rho}, \hat{\sigma}, \hat{\kappa})}^{\hat{\varsigma}} \rightsquigarrow_{\hat{\Sigma}} (e', \hat{\rho}', \hat{\sigma}', \hat{\phi} : \hat{\kappa}), \text{ where}$$

$$\begin{aligned}
\hat{\phi} &= (y, e, \hat{\rho})\\
((\lambda\ (x)\ e'), \hat{\rho}_\lambda) &\in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})\\
\hat{\rho}' &= \hat{\rho}_\lambda[x \mapsto \hat{a}]\\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathcal{A}}(\text{æ}, \hat{\rho}, \hat{\sigma})]\\
\hat{a} &= \widehat{alloc}(x, \hat{\varsigma})
\end{aligned}$$

This is slightly simplified from its analogue in $(\rightsquigarrow_{\tilde{\Sigma}})$. The definitions of e', $\hat{\rho}'$, and $\hat{\sigma}'$ are effectively identical, but the continuation store and continuation address have been replaced with an unbounded stack $\hat{\phi} : \hat{\kappa}$.

Likewise, the return transition also changes to the following.

$$\overbrace{(\text{æ}, \hat{\rho}, \hat{\sigma}, \hat{\phi} : \hat{\kappa})}^{\hat{\varsigma}} \rightsquigarrow_{\hat{\Sigma}} (e, \hat{\rho}', \hat{\sigma}', \hat{\kappa}), \text{ where}$$

$$\begin{aligned}
\hat{\phi} &= (x, e, \hat{\rho}_\kappa)\\
\hat{\rho}' &= \hat{\rho}_\kappa[x \mapsto \hat{a}]\\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathcal{A}}(\text{æ}, \hat{\rho}, \hat{\sigma})]\\
\hat{a} &= \widehat{alloc}(x, \hat{\varsigma})
\end{aligned}$$

To follow a return transition, the stack must contain at least one frame. Then the appropriate e is reinstated with the environment $\hat{\rho}$ extended with an address for x. The store is extended and whatever stack tail existed after $\hat{\phi}$ is the successor's continuation $\hat{\kappa}$.

Unbounded-state injection is defined as we would expect:

$$\hat{\mathcal{I}} : \mathsf{Exp} \to \hat{\Sigma}$$

$$\hat{\mathcal{I}}(e) \triangleq (e, \varnothing, \bot, \epsilon)$$

As before, we lift $(\rightsquigarrow_{\hat{\Sigma}})$ to obtain a monotonic naïve collecting relation $(\rightsquigarrow_{\hat{s}})$ for a program e_0 that is defined over sets of unbounded-states:

$$\hat{s} \in \hat{S} \triangleq \mathcal{P}(\hat{\Sigma})$$

$$\hat{s} \rightsquigarrow_{\hat{s}} \hat{s}' \triangleq \hat{s}' = \{\hat{\varsigma}' \mid \hat{\varsigma} \in \hat{s} \wedge \hat{\varsigma} \rightsquigarrow_{\hat{\Sigma}} \hat{\varsigma}'\} \cup \{\hat{\mathcal{I}}(e_0)\}$$

This analysis is approximate but remains incomputable because the stack can grow without bound. Put another way, the height of the lattice $(\hat{S}, \cup, \cap)$ is infinite and so no finite number of $(\rightsquigarrow_{\hat{s}})$-iterations is guaranteed to obtain a fixed point.

3.2 Store-Widened Unbounded-Stack Analysis

As we will be comparing this unbounded-stack analysis to our new technique using precise store-allocated continuations, we derive a global-store-widened version as before:

$$\begin{aligned}
\hat{\xi} \in \hat{\Xi} &\triangleq \hat{R} \times \widehat{Store} && \text{[state-spaces]}\\
\hat{r} \in \hat{R} &\triangleq \mathcal{P}(\widehat{C}) && \text{[reachable configs.]}\\
\hat{c} \in \hat{C} &\triangleq \mathsf{Exp} \times \widehat{Env} \times \widehat{Kont} && \text{[configurations]}
\end{aligned}$$

A widened transfer function $(\rightsquigarrow_{\hat{\Xi}})$ is defined in terms of $(\rightsquigarrow_{\hat{\Sigma}})$ in exactly the same manner as $(\rightsquigarrow_{\tilde{\Xi}})$ was derived from $(\rightsquigarrow_{\tilde{\Sigma}})$ except that we now have only a single global value store and no continuation store:

$$(\rightsquigarrow_{\hat{\Xi}}) : \hat{\Xi} \to \hat{\Xi}$$

$$(\hat{r}, \hat{\sigma}) \rightsquigarrow_{\hat{\Xi}} (\hat{r}', \hat{\sigma}'), \text{ where}$$

$$\begin{aligned}
\hat{s} &= \{\hat{\varsigma} \mid (e, \hat{\rho}, \hat{\kappa}) \in \hat{r} \wedge (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \rightsquigarrow_{\hat{\Sigma}} \hat{\varsigma}\} \cup \{\hat{\mathcal{I}}(e_0)\}\\
\hat{r}' &= \{(e, \hat{\rho}, \hat{\kappa}) \mid (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \in \hat{s}\}\\
\hat{\sigma}' &= \bigsqcup_{(_,_,\hat{\sigma}'',_) \in \hat{s}} \hat{\sigma}''
\end{aligned}$$

3.3 Pushdown Control-Flow Analysis (PDCFA)

Pushdown control-flow analysis (PDCFA) is a strategy for creating a computable equivalent to the precision of our unbounded-stack machine at a quadratic-factor increase to the complexity class of the underlying finite analysis (e.g., monovariant or 1-call-sensitive) [4]. This strategy tracks both reachable states (or in the store-widened case, configurations) as well as push or pop edges between them. A quadratic blow up comes from the fact that each pair of reachable states may have an explicitly-tracked edge between them. These edges implicitly represent, as possible paths through the graph, the stacks explicitly represented in the unbounded-stack machine. This graph precisely describes the regular expression of all stacks reachable in the pushdown states of the unbounded-stack analysis.

PDCFA formalizes a *Dyke state graph* for this. Where a sequence of pushes may be repeated *ad infinitum*, a Dyke state graph explicitly represents a cycle of push edges and a cycle of pop edges finitely. Broadly speaking, this is also how AAC and our adaptive continuation allocator work, except that such cycles are represented in the store instead of the state graph. A Dyke state graph is a state transition graph where each edge is annotated with either a frame push, a frame pop, or an epsilon. The set of continuations for a particular state in a Dyke state graph is determined by the pushes and pops along the paths that reach that state.

To formalize these Dyke state graphs, we reuse some components of our unbounded-stack machine, continuing to use hats as these machines are closely related:

$$\begin{aligned}
\hat{g} \in \hat{G} &\triangleq \hat{V} \times \hat{E} \times \widehat{Store} && \text{[Dyke graph]}\\
\hat{v} \in \hat{V} &\triangleq \mathcal{P}(Q) && \text{[Dyke vertices]}\\
\hat{q} \in \hat{Q} &\triangleq \mathsf{Exp} \times \widehat{Env} && \text{[Dyke configs.]}\\
\hat{e} \in \hat{E} &\triangleq \mathcal{P}(\hat{Q} \times \widehat{Frame}_{\pm} \times \hat{Q}) && \text{[Dyke edges]}\\
\hat{\phi}_{\pm} \in \widehat{Frame}_{\pm} &\triangleq \widehat{Frame} \times \{\textbf{push}, \textbf{pop}\} && \text{[edge actions]}
\end{aligned}$$

For readability, we style an edge $(\hat{q}, (\hat{\phi}, \textbf{push}), \hat{q}') \in \hat{e}$ like so:

$$\hat{q} \xrightarrow{\hat{\phi}^+} \hat{q}' \in \hat{e}$$

It would be too verbose to formalize all the machinery required to compute a valid Dyke state graph. Instead, we define it from a completed unbounded-stack analysis $\hat{\xi}$. The function $\mathcal{DSG} : \hat{\Xi} \to \hat{G}$ produces a Dyke state graph from a fixed-point $\hat{\xi}$ for $(\rightsquigarrow_{\hat{\Xi}})$. The graph $\hat{g} = \mathcal{DSG}(\hat{\xi})$ is a valid Dyke state graph analysis for a program e_0 when $\hat{\xi}$ is the unbounded-stack analysis of e_0.

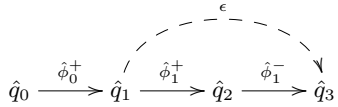
$$\mathcal{DSG}(\overbrace{(\hat{r}, \hat{\sigma})}^{\hat{\xi}}) \triangleq \overbrace{(\hat{v}, \hat{e}, \hat{\sigma})}^{\hat{g}}, \text{ where}$$

$$\begin{aligned}
\hat{v} &= \{(e, \hat{\rho}) \mid (e, \hat{\rho}, \hat{\kappa}) \in \hat{r}\}\\
\hat{e} &= \{(e, \hat{\rho}) \xrightarrow{\hat{\phi}^+} (e', \hat{\rho}') \mid (e, \hat{\rho}, \hat{\kappa}) \in \hat{r}\\
&\qquad \wedge (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \rightsquigarrow_{\hat{\Sigma}} (e', \hat{\rho}', \hat{\sigma}, \hat{\phi} : \hat{\kappa})\}\\
&\cup \{(e, \hat{\rho}) \xrightarrow{\hat{\phi}^-} (e', \hat{\rho}') \mid (e, \hat{\rho}, \hat{\kappa}) \in \hat{r}\\
&\qquad \wedge (e, \hat{\rho}, \hat{\sigma}, \hat{\phi} : \hat{\kappa}) \rightsquigarrow_{\hat{\Sigma}} (e', \hat{\rho}', \hat{\sigma}, \hat{\kappa})\}
\end{aligned}$$

Although we do not formalize transition relations for Dyke state graphs themselves, it will be helpful for us to illustrate the major source of additional complexity in engineering a PDCFA directly. In the finite-state analysis, a transition is able to trivially compute a set of stacks by looking up the current continuation address in the continuation store. In the unbounded-stack analysis, a transition is able to trivially compute the stack by looking at the final component of the state or configuration being transitioned. In a Dyke state graph, canceling sequences of pushes and pops may place the set of topmost stack frames on edges arbitrarily distant from the configuration $\hat{q}$ being transitioned. In this way, the implicitness of stacks in a Dyke state graph obfuscates one of the most common operations needed to compute the analysis (i.e., stack introspection). As an example, observe how the topmost stack frame $\hat{\phi}_0$ for $\hat{q}_3$ is located elsewhere in the graph:

$$\hat{q}_0 \xrightarrow{\hat{\phi}_0^+} \hat{q}_1 \xrightarrow{\hat{\phi}_1^+} \hat{q}_2 \xrightarrow{\hat{\phi}_1^-} \hat{q}_3$$

PDCFA therefore requires a non-trivial algorithm for stack introspection [5] and extra analysis machinery overall. Specifically, PDCFA requires the inductive maintenance of an *epsilon closure graph* in addition to the Dyke state graph as seen in the following.



This structure makes all sequences of canceling stack actions explicit as an epsilon edge. As we will see, this epsilon closure graph represents unnecessary additional complexity for both computer and analysis developer.

3.4 Precise Allocation of Continuations (AAC)

Abstracting abstract control (AAC) [8] is another polynomial-time method for obtaining perfect stack precision. This technique works by store-allocating continuations using addresses unique enough to ensure no spurious merging and, like PDCFA, does not require foreknowledge of the polyvariance (e.g., context sensitivity) being used in the value store. The method is worse than PDCFA's quadratic-factor increase in run-time complexity. In the monovariant and store-widened case, its authors believe it to be in $O(n^8)$ [7]. However, AAC makes perfect stack precision available *for free* in terms of development cost (i.e., labor).

Given the standard finite-state abstraction we built up in section 2.2, we can define AAC's essential strategy in a single line:

$$\widetilde{alloc}_{\kappa\,\text{AAC}}((e, \tilde{\rho}, \tilde{\sigma}, \tilde{\kappa}), e', \tilde{\rho}', \tilde{\sigma}') \triangleq (e', \tilde{\rho}', e, \tilde{\rho}, \tilde{\sigma})$$

That is, continuations are stored at an address unique to the target state's expression e' and environment $\tilde{\rho}'$ as well as the source state's expression e, environment $\tilde{\rho}$, and store $\tilde{\sigma}$.

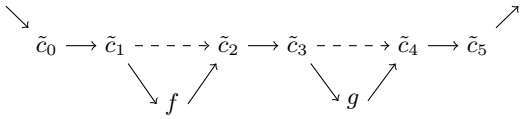
We have simplified AAC slightly and translated its notation to give this definition in the terms of our framework. A more faithful presentation of AAC shows fundamental differences between their framework and ours. AAC uses an eval-apply semantics and explodes each flow set into a set of distinct states across every application. The exact address AAC proposes using is $((\texttt{(}\lambda\ \texttt{(}y\texttt{)}\ e'\texttt{)}, \tilde{\rho}_\lambda), \widetilde{clo}, \tilde{\sigma})$ (Figure 7 in [8]) where $(\texttt{(}\lambda\ \texttt{(}y\texttt{)}\ e'\texttt{)}, \tilde{\rho}_\lambda)$ is the target closure of an application, $\widetilde{clo}$ is one particular abstract closure flowing to y, and $\tilde{\sigma}$ is the value store in the source state. Our components e' and $\tilde{\rho}'$ are isomorphic to the target closure in the sense that e' is identical and $\tilde{\rho}'$ is produced from the combination of $\tilde{\rho}_\lambda$ and y. The source state's components (e, $\tilde{\rho}$, $\tilde{\sigma}$) are not as specific as $\widetilde{clo}$ and $\tilde{\sigma}$, but they do uniquely determine a flow set $\tilde{d}$ (the result of $\tilde{\mathcal{A}}$ invoked on f) that contains $\widetilde{clo}$. However, a semantics using an eval-apply factoring like AAC is needed to obtain a unique continuation address for every closure propagated across an application. This would have significantly complicated our presentation of the finite-state analysis, and in section 4 we will see that being specific to $\widetilde{clo}$ adds run-time complexity to an analysis without adding any precision.

The intuition for AAC is that by allocating continuations specific to both the source state and target state of a call-site transition, no merging may occur when returning according to this (transition-specific) continuation-address. If we were to add some arbitrary additional context sensitivity (e.g., 3-call-sensitivity), this information would be encoded in $\tilde{\rho}'$ and inherited by $\widetilde{alloc}_{\kappa\,\text{AAC}}$ upon producing an address. Including this target-state binding environment in continuation addresses is the key reason why AAC allocates precise continuation addresses.

In section 4, we will see that only the target state's expression e' and environment $\tilde{\rho}'$ are truly necessary for obtaining the perfect stack precision of our unbounded-stack machine. Including components of the transition's source state, its store, or its flow set only adds run-time complexity that is unnecessary for achieving perfect stack precision. This optimization extends AAC's core insight to be computationally *for free* while remaining precise and developmentally *for free*.

4. Perfect Stack Precision for Free

The primary intuition of our work can be illustrated by considering a set of intraprocedural configurations for some function invocation as in the following with $\tilde{c}_0$ through $\tilde{c}_5$.



The configuration $\tilde{c}_0$ represents the entry point to the function, and its incoming edge is a call-site transition. The configuration $\tilde{c}_5$ represents an exit point for the function, and its outgoing edge is a return-point transition. A transition where one intraprocedural configuration follows another, like $\tilde{c}_0 \rightarrow \tilde{c}_1$, is not technically possible in our restricted ANF language but in more general languages would be. The function's body may call other functions f and g whose configurations are not a part of the same intraprocedural set of nodes. The primary insight behind our technique is that *a set of intraprocedural configurations (like $\tilde{c}_0$ through $\tilde{c}_5$) necessarily share the exact same set of genuine continuations* (in this example, the incoming call-sites for $\tilde{c}_0$).

We call the set of configurations $\tilde{c}_0$ through $\tilde{c}_5$ an *intraprocedural group* because they are those configurations that represent the body of a function for a single abstract invocation—defined by an entry point unique to some e and $\tilde{\rho}$. Our central insight is to notice that this idea of an intraprocedural group also corresponds to those configurations that share a single set of continuations. Our finite-state machine represents this set of continuations with a continuation address, so if this continuation address is precise enough to uniquely determine an intraprocedural group's entry point (e and $\tilde{\rho}$), then it can be used for all configurations in that same group. Thus our allocator may be defined as simply:

$$\widetilde{alloc}_{\kappa\,\text{P4F}}((e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa), e', \tilde{\rho}', \tilde{\sigma}') = (e', \tilde{\rho}')$$

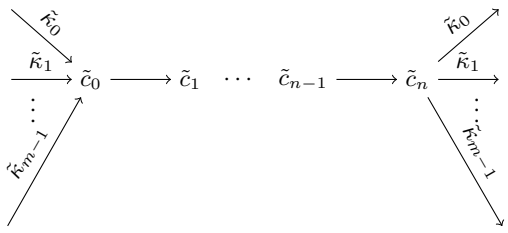
The impact of this change is easily missed, belied by its simplicity. We allocate a continuation based only on the expression and environment at the entry point of each intraprocedural sequence of `let`-forms and it is precisely reinstated when each of the calls in these `let`-forms return.

Recall that the monovariant continuation allocator in our example from section 2.5 resulted in return-flow merging because a single continuation address was being used for transitions to multiple entry points of different intraprocedural groups. More generally, return-flow merging occurs in a finite-state analysis when, at some return-point configuration $(\text{æ}, \tilde{\rho}_{\text{æ}}, \tilde{a}_\kappa)$, the set of continuations for $\tilde{a}_\kappa$ is less precise than the set of source configurations that transition to the entry point $(e, \tilde{\rho})$ of the same intraprocedural group. Because we allocate a continuation address specific to this exact entry point, and because that address is propagated by shallowly copying it to each return point for the same intraprocedural group, the set of continuations will be as precise as the set of source configurations transitioning to the same entry point in all cases. This means the return-flow merging problem cannot occur when using $\widetilde{alloc}_{\kappa\,\text{P4F}}$ and neither is there a run-time overhead for stack introspection.

In section 5, we formalize these intuitions and provide a proof that our unbounded-stack analysis simulates (i.e., is no more precise than) a finite-state analysis when using $\widetilde{alloc}_{\kappa\,\text{P4F}}$.

4.1 Complexity

To see why this allocation scheme leads to only a constant-factor overhead, consider a set of configurations $\tilde{c}_0, \tilde{c}_1, \cdots, \tilde{c}_n$ that form an intraprocedural group and a set of call sites transitioning to $\tilde{c}_0$ with the continuations $\tilde{\kappa}_0, \tilde{\kappa}_1, \cdots, \tilde{\kappa}_{m-1}$. We can diagrammatically visualize this as the following.



Note that, for each call site, there is a corresponding return flow using the same continuation. Our allocation strategy means that all of the configurations $\tilde{c}_0, \tilde{c}_1, \cdots, \tilde{c}_n$ use the same continuation address $(e, \tilde{\rho})$. The global continuation store then maps this address to the set $\{\tilde{\kappa}_0, \tilde{\kappa}_1, \cdots, \tilde{\kappa}_{m-1}\}$.

Now consider what must be done if a new call site transitions to c_0. First, the continuation store must be extended to contain the continuation for this new call site, say $\tilde{\kappa}_m$, in the continuation set at the address $(e, \tilde{\rho})$. Then the corresponding return edge transitions must be added. Note that none of $\tilde{c}_0, \tilde{c}_1, \cdots, \tilde{c}_{n-1}$ need to be modified or accessed. The only work done here beyond that of the underlying analysis is the extension of the continuation store by adding $\tilde{\kappa}_m$ at $(e, \tilde{\rho})$ and the addition of a corresponding return edge at return points. Thus, the additional work is a constant factor of the number of times a continuation is added to the continuation store.

A naïve analysis might lead us to conclude that this is bounded by the product of the number of continuation addresses and the number of continuations. However, there is a tighter bound. Each transition adds only one continuation to the continuation store. Thus the work done is a constant factor of the number of transitions in the underlying analysis.

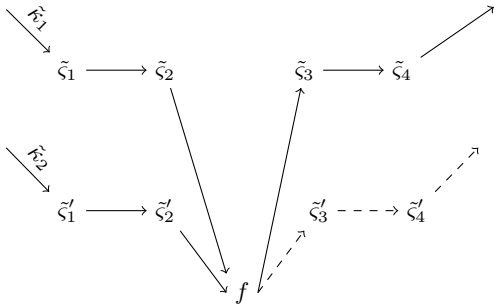
Note that this differs from AAC, which may make duplicate copies of the continuation set for an intraprocedural group as it produces one for each combination of components e, $\tilde{\rho}$, and $\tilde{\sigma}$ drawn from the source states transitioning to it. As a consequence, AAC allocates addresses strictly more unique than the target $(e', \tilde{\rho}')$ configuration. Two different source expressions e_0 and e_1 may both have transitions to $(e', \tilde{\rho}')$, but AAC will produce two different target configurations $\tilde{c}'_0$ and $\tilde{c}'_1$ because the continuation addresses they allocate will be distinct. This difference is maintained through the two variants of the function starting at e' with environment $\tilde{\rho}'$, and when an exit point æ is reached for each, the expression and its environment are the same and propagate the same values to two sets of continuations. Thus, these continuation addresses and the sets of stacks they represent are kept separate without any benefit.

PDCFA, on the other hand, is more complex for an entirely different reason: the epsilon closure graph. Without the epsilon closure graph, PDCFA has no way to efficiently determine a topmost stack frame at each return transition. Both our method and AAC's method make this trivial by propagating an address explicitly to each state. While our method allows a continuation address to be shallowly propagated across each intraprocedural node in a function, the epsilon closure graph recomputes a separate set of incoming epsilon edges for every node. This means that the number of such edges for a given entry point $(e, \hat{\rho})$ is the number of callers times the number of intraprocedural nodes. This is a quadratic blow up from the number of nodes in a finite-state model. This is why monovariant store-widened PDCFA is in $O(n^6)$ instead of in $O(n^3)$ like traditional 0-CFA. We are able to naturally exploit our insight that each intraprocedural node following an entry point $(e, \hat{\rho})$ shares the same set of continuations (i.e., the same epsilon edges) by propagating a pointer to this set instead of rebuilding it for each node. PDCFA is unable to exploit this insight without adding machinery to propagate only a shallow copy of an incoming epsilon edge set intraprocedurally. It is likely that this insight could also be imported into the PDCFA style of analysis to yield a variant of PDCFA that incurs only a constant-factor overhead, but this would require additional machinery.

4.2 Constant Overhead Requires Store Widening

That no function can have two entry points that lead to the same exit point is a genuine restriction worth discussing further. If this were not true, our technique would be precise (assuming multiple entry points are not merged), but it would not necessarily be a constant-factor increase in complexity. The combination of no store widening (per-state value stores) and mutation is a good example of how this situation could arise.

To see how per-state stores can cause a further blow up in complexity, consider a function that is called with two different continuations and two different stores. Without store widening, each store causes a different state to be created for the entry point $(e, \tilde{\rho})$ of the function. In the following diagram for example, $\tilde{\varsigma}_1$ is the state for the entry point with one store and $\tilde{\varsigma}'_1$ the state for the entry point with another store.



Now suppose that along both sequences of states there is a call to some function f and that f contains a side effect that causes the previously different value stores to become equal. For example, in $\tilde{\varsigma}_1$ perhaps the address for x maps to $\{\texttt{\#t}\}$ and in $\tilde{\varsigma}'_1$ it maps to $\{\texttt{\#f}\}$. If x becomes bound to $\{\texttt{\#t}, \texttt{\#f}\}$ along both paths in the body of f, the stores along both paths would become identical.

A problem now arises. Should f return only to one state using this common store such as $\tilde{\varsigma}_3$ or should it return to two different states (with identical stores) such as both $\tilde{\varsigma}_3$ and $\tilde{\varsigma}'_3$? Either choice has drawbacks. The semantics we have given would naturally yield the latter option, producing two distinct states that differ only by their continuation addresses (their original entry point). Because these states are otherwise identical, splitting $\tilde{\kappa}_1$ and $\tilde{\kappa}_2$ into sets represented by two different continuation addresses results in additional transitions and complexity without any benefit. Arguably, these continuation sets should be merged and represented by a single address. This corresponds to the former option and could save on run-time complexity but only at the cost of additional analysis machinery. This means per-state stores are incompatible with our goal of obtaining perfect stack precision *for free* in both senses (running time and human labor).

4.3 Implementation

We have implemented both our technique and AAC's technique for analysis of a simplified Scheme intermediate language. This language extends Exp with a variety of additional core forms including conditionals, mutation, recursive binding, tail calls, and a library of primitive operations. Our implementation was written in Scala and executed using Scala 2.11 for OSX on an Intel Core i5 (1.3 GHz) with 4GB of RAM. It is built upon the implementation of Earl et al. [5], which implements both traditional k-CFA and PDCFA. The test cases we ran came from the Larceny R6RS benchmark suite (ack, cpstak, tak) and examples compiled from the previous litera-

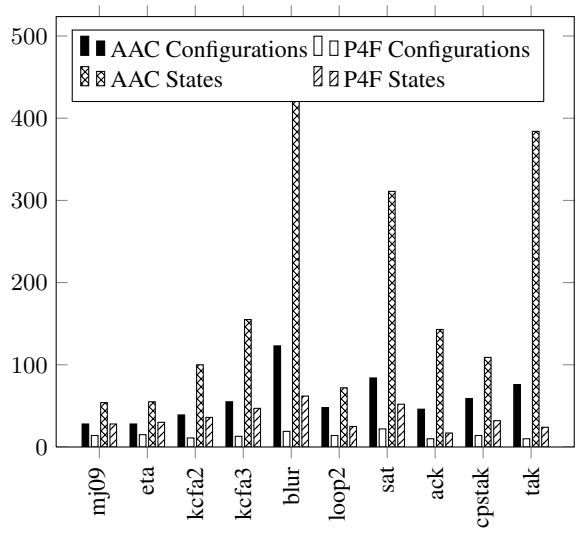


Figure 2. A monovariant comparison.

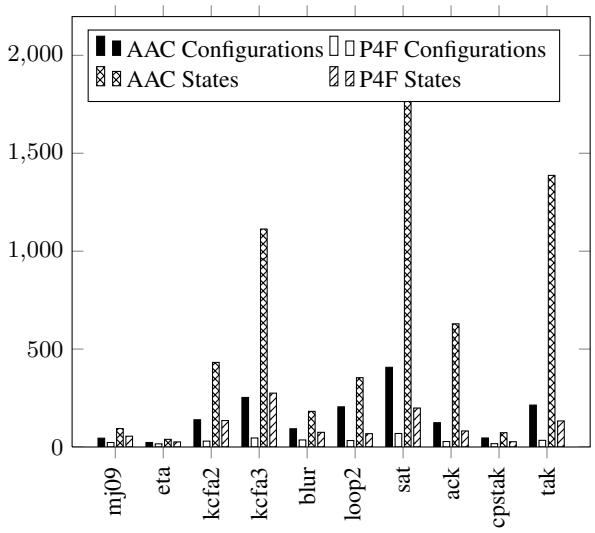


Figure 3. A 1-call-sensitive comparison.

ture on obtaining perfect stack precision (mj09, eta, kcfa2, kcfa3, blur, loop2, sat). As a sanity check, we have verified that both AAC and our method produce results of equivalent precision in every case. We ran each comparison using both a monovariant value-store allocator (figure 2), and a 1-call-sensitive polyvariant allocator (figure 3). Across the board, our method requires visiting strictly fewer machine configurations. In some of these cases the difference is rather small, but in others it is significant. We saw as much as a $16.0\times$ improvement in the monovariant analysis and as much as a $10.4\times$ improvement in the context-sensitive analysis. The mean speedup in terms of states visited was $5.4\times$ and $4.9\times$ in the monovariant and context-sensitive analyses, respectively.

5. Proof of Precision

Proving soundness is fairly straight forward as discussed in section 2.3, but proving precision poses a greater challenge. To do this, we first define a simulation relation ($\sqsupseteq_\equiv$) where $\hat{\xi} \sqsupseteq_\equiv \tilde{\xi}$ (read as "$\hat{\xi}$ simulates $\tilde{\xi}$") if and only if all stored values and machine configurations in $\tilde{\xi}$ (including stacks implicit in this configuration) are accounted for in the unbounded-stack representation $\hat{\xi}$. Usually, the next step in such a proof would be to show that taking parallel steps preserves precision as in fallacy 1.

Fallacy 1 (Steps preserve precision)**.** *If $\hat{\xi} \rightsquigarrow_{\hat{\Sigma}} \hat{\xi}'$ and $\tilde{\xi} \rightsquigarrow_{\tilde{\Sigma}} \tilde{\xi}'$, then $\hat{\xi} \sqsupseteq_\equiv \tilde{\xi}$ implies $\hat{\xi}' \sqsupseteq_\equiv \tilde{\xi}'$.*

However, fallacy 1 is not true. This is because after some finite number of steps $\tilde{\xi}$ may contain a cycle in its continuation store. This means that an infinite family of successively longer stacks must also be in $\hat{\xi}$ for precision to hold. After a finite number of steps, however, all stacks in $\hat{\xi}$ are bounded by a finite length. Hence, there are stacks that precision says should be in $\hat{\xi}$ that are not.

We thus take a different approach to proving precision. Before going into the details, the high-level overview of this proof is as follows. Instead of stepping both $\hat{\xi}$ and $\tilde{\xi}$ in parallel, we show that successive steps of $\tilde{\xi}$ are all precise relative to any $\hat{\xi}$ that is already at a fixed point (i.e., theorem 11 found at the end of this section). To show this, we need two inductions. One is over the steps taken by $\tilde{\xi}$, and the other is over the stacks implied by $\tilde{\xi}$. To separate these inductions, we define a well-formedness property (*wf* in figure 7) that we can show is preserved by iterative steps from an initial $\tilde{\xi}_0$ (lemma 5) and for which we can show that any well-formed $\tilde{\xi}$ is precise relative to any $\hat{\xi}$ that is at a fixed point (lemmas 9 and 10).

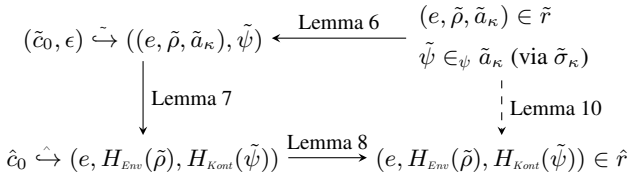


Figure 4. The logical chain proving lemma 10. Assumes $(\hat{r}, \hat{\sigma})$ is at a fixed point and $(\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa)$ is well-formed.

The well-formedness property is defined in terms of two additional concepts. First, we formally define the stacks, $\tilde{\psi}$, implied by a continuation address, $\tilde{a}_\kappa$, and continuation store, $\tilde{\sigma}_\kappa$, in terms of a relation $\tilde{\psi} \in_\psi \tilde{a}_\kappa$ (via $\tilde{\sigma}_\kappa$) that we define. Then, we define paths, $(\hat{\hookrightarrow})$ and $(\tilde{\hookrightarrow})$, through $\hat{\xi}$ and $\tilde{\xi}$ in terms of a sequence of state steps, $(\rightsquigarrow_{\hat{\Sigma}})$ and $(\rightsquigarrow_{\tilde{\Sigma}})$, between states represented by configurations in $\hat{\xi}$ and $\tilde{\xi}$. This allows us to prove the precision of any well-formed $\tilde{\xi}$ (i.e., lemma 10) through a logical chain informally shown in figure 4. In lemma 6, we show that for any configuration $(e, \tilde{\rho}, \tilde{a}_\kappa)$ in the $\tilde{r}$ of a $\tilde{\xi}$ and any $\tilde{\psi}$ implied by a_κ with the continuation store σ_κ of $\tilde{\xi}$, there exists a path from the initial configuration $\tilde{c}_0$ with an empty stack ϵ to the configuration $(e, \tilde{\rho}, \tilde{a}_\kappa)$ with the implied stack $\tilde{\psi}$. In lemma 7, we then show that there exists a corresponding path in $\hat{\xi}$ from $\hat{c}_0$ to $(e, H_{Env}(\tilde{\rho}), H_{Kont}(\tilde{\psi}))$. Finally, in lemma 8, we show that the endpoint of that path is in $\hat{\xi}$ and thus the set of reachable configurations in $\tilde{\xi}$ is precise relative to $\hat{\xi}$ (lemma 10).

5.1 Definitions and Assumptions

In order to prove precision, we first require that the address spaces for both $\hat{\sigma}$ and $\tilde{\sigma}$ correspond as follows.

Assumption 2 (Address equivalence)**.** *There exists an equivalence ($\equiv_{Addr}$) between finite-state-machine addresses ($\widehat{Addr}$) and unbounded-stack-machine addresses ($\widetilde{Addr}$) that can be decomposed into a bijection $\widehat{Addr} \underset{H_{Addr}}{\overset{T_{Addr}}{\rightleftharpoons}} \widetilde{Addr}$.*

$$((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \tilde{\hookrightarrow} ((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \text{ (via } (\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)), \text{where}$$
$$(e,\tilde{\rho},\tilde{a}_\kappa) \in \tilde{r}' \qquad \tilde{\psi} \in_\psi \tilde{a}_\kappa \text{ (via } \tilde{\sigma}'_\kappa)$$

$$((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \tilde{\hookrightarrow} ((e',\tilde{\rho}'_\kappa,\tilde{a}'_\kappa),\tilde{\psi}') \text{ (via } (\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)), \text{where}$$
$$\tilde{\rho}'_\kappa = \tilde{\rho}_\kappa[x \mapsto \widetilde{alloc}(x,(\text{æ},\tilde{\rho}'',\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}''_\kappa))]$$
$$(\text{æ},\tilde{\rho}'',\tilde{a}''_\kappa) \in \tilde{r} \qquad (e',\tilde{\rho}'_\kappa,\tilde{a}'_\kappa) \in \tilde{r}' \qquad ((x,e',\tilde{\rho}_\kappa),\tilde{a}'_\kappa) \in \tilde{\sigma}_\kappa(\tilde{a}''_\kappa)$$
$$((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \tilde{\hookrightarrow} ((\text{æ},\tilde{\rho}'',\tilde{a}''_\kappa),((x,e',\tilde{\rho}_\kappa),\tilde{a}'_\kappa):\tilde{\psi}') \text{ (via } (\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa))$$
$$(\text{æ},\tilde{\rho}'',\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}''_\kappa) \overset{\sqsubseteq}{\tilde{\rightsquigarrow}}_{\tilde{\Sigma}} (e',\tilde{\rho}'_\kappa,\tilde{\sigma}',\tilde{\sigma}'_\kappa,\tilde{a}'_\kappa)$$

$$((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \tilde{\hookrightarrow} ((e',\tilde{\rho}',\tilde{a}'_\kappa),((x,e'',\tilde{\rho}''),\tilde{a}''_\kappa):\tilde{\psi}') \text{ (via } (\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)), \text{where}$$
$$((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \in \tilde{r} \qquad ((x,e'',\tilde{\rho}''),\tilde{a}''_\kappa) \in \tilde{\sigma}'_\kappa(\tilde{a}'_\kappa)$$
$$((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \tilde{\hookrightarrow} ((\texttt{(let ([}x\ (f\ \text{æ})\texttt{])}\ e''),\tilde{\rho}'',\tilde{a}''_\kappa),\tilde{\psi}') \text{ (via } (\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa))$$
$$(\texttt{(let ([}x\ (f\ \text{æ})\texttt{])}\ e''),\tilde{\rho}'',\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}''_\kappa) \overset{\sqsubseteq}{\tilde{\rightsquigarrow}}_{\tilde{\Sigma}} (e',\tilde{\rho}',\tilde{\sigma}',\tilde{\sigma}'_\kappa,\tilde{a}'_\kappa)$$

Figure 5. Finite-state paths

From this we can build up equivalences ($\equiv_{Env}$, $\equiv_{Frame}$, $\equiv_{Clo}$) and precision relations ($\sqsupseteq_\Xi$, $\sqsupseteq_{Store}$, $\sqsupseteq_R$, $\sqsupseteq_D$, $\sqsupseteq_{Store}$) for all the components of our machine. In addition, we can define conversion bijections (H_{Env}, T_{Env}, H_{Frame}, T_{Frame}, H_{Clo}, T_{Clo}, H_D, T_D, H_{Store}, T_{Store}) for most but not all of the components. These relations have the following signatures.

$$(\sqsupseteq_\Xi) \subseteq \hat{\Xi} \times \tilde{\Xi} \quad \text{[state-space precision]}$$
$$(\sqsupseteq_{Store}) \subseteq \widehat{Store} \times \widetilde{Store} \quad \text{[store precision]}$$
$$(\sqsupseteq_R) \subseteq \hat{R} \times \tilde{R} \times \widetilde{KStore} \quad \text{[reachable configs. precision]}$$
$$(\equiv_{Env}) \subseteq \widehat{Env} \times \widetilde{Env} \quad \text{[env. equivalence]}$$
$$(\equiv_{Frame}) \subseteq \widehat{Frame} \times \widetilde{Frame} \quad \text{[frame equivalence]}$$
$$(\sqsupseteq_D) \subseteq \hat{D} \times \tilde{D} \quad \text{[flow-set precision]}$$
$$(\equiv_{Clo}) \subseteq \widehat{Clo} \times \widetilde{Clo} \quad \text{[closure equivalence]}$$

In addition, with the following assumption, we require that the value allocators respect the address correspondence.

Assumption 3 (Allocation equivalence)**.** *If $\hat{\rho} \equiv_{Env} \tilde{\rho}$, $\hat{\sigma} \sqsupseteq_{Store} \tilde{\sigma}$, and $\tilde{\psi} \in_\psi \tilde{a}_\kappa$ (via $\tilde{\sigma}_\kappa$), then:*

$$\widehat{alloc}(x,(e,\hat{\rho},\hat{\sigma},H_{Kont}(\tilde{\psi}))) \equiv_{Addr} \widetilde{alloc}(x,(e,\tilde{\rho},\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa))$$

This assumption uses ($\in_\psi$) and H_{Kont}, which deal with the stacks implied by an address and continuation store. We define an implied stack as an unbounded list of finite-state continuations $\tilde{\kappa}$:

$$\tilde{\psi} \in \tilde{\Psi} = \widetilde{Kont}^* \quad \text{[implied stack]}$$

These $\tilde{\psi}$ are an intermediate representation in that, like $\hat{\kappa}$, their structure is unbounded, but each element is taken directly from the finite-state machine. We define a trinary relation ($\in_\psi$) that specifies which $\tilde{\psi}$ are implied by an $\tilde{a}_\kappa$ in $\tilde{\sigma}_\kappa$. This has the following base case and inductive case:

$$\epsilon \in_\psi \tilde{a}_{\text{halt}} \text{ (via } \tilde{\sigma}_\kappa)$$

$$(e,\hat{\rho},\hat{\kappa}) \hat{\hookrightarrow} (e,\hat{\rho},\hat{\kappa}) \text{ (via } \hat{r},\hat{\sigma})$$

$$(e,\hat{\rho},\hat{\kappa}) \hat{\hookrightarrow} (e',\hat{\rho}',\hat{\kappa}') \text{ (via } \hat{r},\hat{\sigma}), \text{where}$$
$$(e,\hat{\rho},\hat{\kappa}) \hat{\hookrightarrow} (e'',\hat{\rho}'',\hat{\kappa}'') \text{ (via } \hat{r},\hat{\sigma})$$
$$(e'',\hat{\rho}'',\hat{\sigma},\hat{\kappa}'') \overset{\sqsubseteq}{\hat{\rightsquigarrow}}_{\hat{\Sigma}} (e',\hat{\rho}',\hat{\sigma},\hat{\kappa}')$$

Figure 6. Unbounded-stack paths

$$(\tilde{\phi},\tilde{a}'_\kappa) \in \tilde{\sigma}_\kappa(\tilde{a}_\kappa) \wedge \tilde{\psi} \in_\psi \tilde{a}'_\kappa \text{ (via } \tilde{\sigma}_\kappa) \wedge \tilde{a}_\kappa \neq \tilde{a}_{\text{halt}}$$
$$\implies ((\tilde{\phi},\tilde{a}'_\kappa):\tilde{\psi}) \in_\psi \tilde{a}_\kappa \text{ (via } \tilde{\sigma}_\kappa)$$

Then given such a $\tilde{\psi}$, we can directly construct its equivalent unbounded stack:

$$H_{Kont}(\epsilon) \triangleq \epsilon \qquad H_{Kont}((\tilde{\phi},\tilde{a}_\kappa):\tilde{\psi}) \triangleq H_{Frame}(\tilde{\phi}) : H_{Kont}(\tilde{\psi})$$

Also, given a finite-state configuration $\tilde{c}$ and an implicit stack $\tilde{\psi}$, we can construct an unbounded-stack configuration $\hat{c}$:

$$H_C((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \triangleq (e,H_{Env}(\tilde{\rho}),H_{Kont}(\tilde{\psi}))$$

Next, in figures 5 and 6 , we define paths to configurations. For ($\hat{\hookrightarrow}$) this is defined by a base case from a configuration to itself and a recursive case that builds onto an existing path with a step within $\hat{\xi}$. This uses a variation of the step relation defined in figure 8 that allows the output store of the step to be a sub-store of the store in $\hat{\xi}$. The ($\tilde{\hookrightarrow}$) relation is similar but adds extra side conditions that ensure invariants used in our proof.

Then, in figure 7, we define well-formedness. This is a binary predicate with the first argument $\tilde{\xi}$ being the predecessor of the second argument $\tilde{\xi}'$, which is the result we say is well-formed. This predicate is defined in terms of several sub-properties. The $wf_{\tilde{\xi}}$ property requires $\tilde{\xi}$ be well-formed and the predecessor of $\tilde{\xi}'$.

The $wf_\sqsubseteq$ property requires that $\tilde{\xi}'$ be component-wise greater than or equal to $\tilde{\xi}$. The wf_{init} and wf_{halt} properties respectively

$$wf(\tilde{\xi},\tilde{\xi}') \triangleq wf_{\tilde{\xi}}(\tilde{\xi},\tilde{\xi}') \wedge wf_{\sqsubseteq}(\tilde{\xi},\tilde{\xi}') \wedge wf_{\text{init}}(\tilde{\xi},\tilde{\xi}') \wedge wf_{\text{halt}}(\tilde{\xi},\tilde{\xi}') \wedge wf_{\tilde{r}}(\tilde{\xi},\tilde{\xi}') \wedge wf_{\tilde{\sigma}}(\tilde{\xi},\tilde{\xi}') \wedge wf_{\tilde{\sigma}_\kappa}(\tilde{\xi},\tilde{\xi}')$$

$$wf_{\tilde{\xi}}(\tilde{\xi},\tilde{\xi}') \triangleq (\tilde{\xi} = \tilde{\xi}' = (\{(e_0,\varnothing,\tilde{a}_{\text{halt}})\},\bot,\bot)) \vee (\tilde{\xi} \rightsquigarrow_{\tilde{\Xi}} \tilde{\xi}' \wedge \exists \tilde{\xi}''.\ wf(\tilde{\xi}'',\tilde{\xi}))$$

$$wf_{\sqsubseteq}((\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)) \triangleq (\tilde{r} \subseteq \tilde{r}') \wedge (\tilde{\sigma} \sqsubseteq \tilde{\sigma}') \wedge (\tilde{\sigma}_\kappa \sqsubseteq \tilde{\sigma}'_\kappa)$$

$$wf_{\text{init}}(\tilde{\xi},(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)) \triangleq (e_0,\varnothing,\tilde{a}_{\text{halt}}) \in \tilde{r}'$$

$$wf_{\text{halt}}(\tilde{\xi},(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)) \triangleq \forall \tilde{\kappa}.\ \tilde{\kappa} \notin \tilde{\sigma}'_\kappa(\tilde{a}_{\text{halt}})$$

$$wf_{\tilde{r}}(\tilde{\xi},(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)) \triangleq \forall (e,\tilde{\rho},\tilde{a}_\kappa) \in \tilde{r}'.\ \exists \tilde{\psi} \in_\psi \tilde{a}_\kappa \text{ (via } \tilde{\sigma}'_\kappa).\ ((e_0,\varnothing,\tilde{a}_{\text{halt}}),\epsilon) \tilde{\hookrightarrow} ((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \text{ (via } \tilde{\xi},(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa))$$

$$wf_{\tilde{\sigma}}((\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)) \triangleq \forall \tilde{a}.\ \forall \widetilde{clo} \in \tilde{\sigma}'(\tilde{a}).\ \exists (e',\tilde{\rho}',\tilde{a}'_\kappa) \in \tilde{r}'.\ (e,\tilde{\rho},\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa) \overset{\sqsubseteq}{\rightsquigarrow}_{\tilde{\Sigma}} (e',\tilde{\rho}',\tilde{\sigma}',\tilde{\sigma}'_\kappa,\tilde{a}'_\kappa) \text{ (with } \tilde{a},\widetilde{clo})$$

$$wf_{\tilde{\sigma}_\kappa}((\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)) \triangleq \forall \tilde{a}'_\kappa.\ \forall ((x,e,\tilde{\rho}_\kappa),\tilde{a}_\kappa) \in \tilde{\sigma}'_\kappa(\tilde{a}'_\kappa).\ \exists e'_\kappa.\ \exists \tilde{\rho}'_\kappa.\ \tilde{a}'_\kappa = (e'_\kappa,\tilde{\rho}'_\kappa) \wedge Entry(\tilde{a}'_\kappa) \in \tilde{r}' \wedge \exists f.\ \exists \text{æ}.\ (\texttt{(let ([}x\ (f\ \text{æ})\texttt{]) }e),\tilde{\rho}_\kappa,\tilde{a}_\kappa) \in \tilde{r} \wedge (\texttt{(let ([}x\ (f\ \text{æ})\texttt{]) }e),\tilde{\rho}_\kappa,\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa) \overset{\sqsubseteq}{\rightsquigarrow}_{\tilde{\Sigma}} (e'_\kappa,\tilde{\rho}'_\kappa,\tilde{\sigma}',\tilde{\sigma}'_\kappa,\tilde{a}'_\kappa) \text{ (with } \tilde{a}'_\kappa,((x,e,\tilde{\rho}_\kappa),\tilde{a}_\kappa))$$

$$Entry : \widetilde{Addr} \to \tilde{C}$$
$$Entry(\tilde{a}_{\text{halt}}) \triangleq (e_0,\varnothing,\tilde{a}_{\text{halt}})$$
$$Entry((e_\kappa,\tilde{\rho}_\kappa)) \triangleq (e_\kappa,\tilde{\rho}_\kappa,(e_\kappa,\tilde{\rho}_\kappa))$$

Figure 7. Well-formedness properties

require that the initial configuration be in $\tilde{\xi}'$ and that the halt-continuation address a_{halt} not have any continuations associated with it. Finally, $wf_{\tilde{r}}$, $wf_{\tilde{\sigma}}$, and $wf_{\tilde{\sigma}_\kappa}$ ensure that everything in the $\tilde{r}$, $\tilde{\sigma}$, and $\tilde{\sigma}_\kappa$ for $\tilde{\xi}'$ has a reason to be there. For $wf_{\tilde{r}}$, this means that every element of $\tilde{r}$ has some path leading to it. For $wf_{\tilde{\sigma}}$ and $wf_{\tilde{\sigma}_\kappa}$, this means that every value stored in $\tilde{\sigma}$ or $\tilde{\sigma}_\kappa$ has some step ($\rightsquigarrow_{\tilde{\Sigma}}$) that put it there. These are defined in terms of the variations of ($\overset{\sqsubseteq}{\rightsquigarrow}_{\tilde{\Sigma}}$) in figure 8 that have side conditions about the contents of the store.

For $wf_{\tilde{\sigma}_\kappa}$, we define $Entry$, which maps a continuation address to the configuration that is the entry point for the function invocation that contains the configurations using that continuation address.

Finally, with the following assumption, we require that once allocation creates an address it must always produce the same address for the same configuration even if the value or continuation stores have changed.

$$(e,\hat{\rho},\hat{\sigma},\hat{\kappa}) \overset{\sqsubseteq}{\rightsquigarrow}_{\hat{\Sigma}} (e',\hat{\rho}',\hat{\sigma}',\hat{\kappa}'), \text{where } (e,\hat{\rho},\hat{\sigma},\hat{\kappa}) \rightsquigarrow_{\hat{\Sigma}} (e',\hat{\rho}',\hat{\sigma}'',\hat{\kappa}') \quad \hat{\sigma}'' \sqsubseteq \hat{\sigma}'$$

$$(e,\tilde{\rho},\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa) \overset{\sqsubseteq}{\rightsquigarrow}_{\tilde{\Sigma}} (e',\tilde{\rho}',\tilde{\sigma}',\tilde{\sigma}'_\kappa,\tilde{a}'_\kappa), \text{where } (e,\tilde{\rho},\tilde{\sigma}'',\tilde{\sigma}''_\kappa,\tilde{a}_\kappa) \rightsquigarrow_{\tilde{\Sigma}} (e',\tilde{\rho}',\tilde{\sigma}''',\tilde{\sigma}'''_\kappa,\tilde{a}'_\kappa) \quad (\tilde{\sigma}'' \sqsubseteq \tilde{\sigma}) \wedge (\tilde{\sigma}''_\kappa \sqsubseteq \tilde{\sigma}_\kappa) \wedge (\tilde{\sigma}''' \sqsubseteq \tilde{\sigma}') \wedge (\tilde{\sigma}'''_\kappa \sqsubseteq \tilde{\sigma}'_\kappa)$$

$$(e,\tilde{\rho},\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa) \overset{\sqsubseteq}{\rightsquigarrow}_{\tilde{\Sigma}} (e',\tilde{\rho}',\tilde{\sigma}',\tilde{\sigma}'_\kappa,\tilde{a}'_\kappa) \text{ (with } \widetilde{clo}), \text{where } (e,\tilde{\rho},\tilde{\sigma}'',\tilde{\sigma}''_\kappa,\tilde{a}_\kappa) \rightsquigarrow_{\tilde{\Sigma}} (e',\tilde{\rho}',\tilde{\sigma}''',\tilde{\sigma}'''_\kappa,\tilde{a}'_\kappa) \quad (\tilde{\sigma}'' \sqsubseteq \tilde{\sigma}) \wedge (\tilde{\sigma}''_\kappa \sqsubseteq \tilde{\sigma}_\kappa) \wedge (\tilde{\sigma}''' \sqsubseteq \tilde{\sigma}') \wedge (\tilde{\sigma}'''_\kappa \sqsubseteq \tilde{\sigma}'_\kappa) \wedge \widetilde{clo} \in \tilde{\sigma}'''(\tilde{a})$$

$$(e,\tilde{\rho},\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa) \overset{\sqsubseteq}{\rightsquigarrow}_{\tilde{\Sigma}} (e',\tilde{\rho}',\tilde{\sigma}',\tilde{\sigma}'_\kappa,\tilde{a}'_\kappa) \text{ (with } \tilde{\kappa},\tilde{a}''_\kappa), \text{where } (e,\tilde{\rho},\tilde{\sigma}'',\tilde{\sigma}''_\kappa,\tilde{a}_\kappa) \rightsquigarrow_{\tilde{\Sigma}} (e',\tilde{\rho}',\tilde{\sigma}''',\tilde{\sigma}'''_\kappa,\tilde{a}'_\kappa) \quad (\tilde{\sigma}'' \sqsubseteq \tilde{\sigma}) \wedge (\tilde{\sigma}''_\kappa \sqsubseteq \tilde{\sigma}_\kappa) \wedge (\tilde{\sigma}''' \sqsubseteq \tilde{\sigma}') \wedge (\tilde{\sigma}'''_\kappa \sqsubseteq \tilde{\sigma}'_\kappa) \wedge \tilde{\kappa} \in \tilde{\sigma}'''_\kappa(\tilde{a}''_\kappa)$$

Figure 8. Sub-step Relations

Assumption 4 (Allocation consistency)**.** *If $wf(\tilde{\xi},(\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa))$, and the state step $(e,\tilde{\rho},\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa) \rightsquigarrow_{\tilde{\Sigma}} (e',\tilde{\rho}'[x \mapsto \tilde{a}],\tilde{\sigma}'',\tilde{\sigma}''_\kappa,\tilde{a}'_\kappa)$ holds where $\tilde{a} = \widetilde{alloc}(x,(e,\tilde{\rho},\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa))$ and there is a result step $(\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa) \rightsquigarrow_{\tilde{\Xi}} (\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)$, then the corresponding allocation for e, $\tilde{\rho}$, and $\tilde{a}_\kappa$, but with $\tilde{\sigma}'$ and $\tilde{\sigma}'_\kappa$, is the same:*

$$\widetilde{alloc}(x,(e,\tilde{\rho},\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa)) = \widetilde{alloc}(x,(e,\tilde{\rho},\tilde{\sigma}',\tilde{\sigma}'_\kappa,\tilde{a}_\kappa))$$

5.2 Lemmas and Theorems

To start, lemma 5 shows that iterated steps produce well-formed $\tilde{\xi}$.

Lemma 5 (Well-formedness of analysis results)**.** *If $\tilde{\xi}'$ is the result of taking zero or more steps of ($\rightsquigarrow_{\tilde{\Xi}}$), starting from the initial result, $(\{(e_0,\varnothing,\tilde{a}_{halt})\},\bot,\bot)$, then $wf(\tilde{\xi},\tilde{\xi}')$ for some $\tilde{\xi}$.*

Proof. We induct over ($\rightsquigarrow_{\tilde{\Xi}}$) steps. In the base case, we can easily show that the initial result is well-formed. We can also show that for any $\tilde{\xi}' \rightsquigarrow_{\tilde{\Xi}} \tilde{\xi}''$, if $wf(\tilde{\xi},\tilde{\xi}')$ then $wf(\tilde{\xi}',\tilde{\xi}'')$. This is done using sublemmas for the components of well-formedness. We omit these for space. □

Next, with lemma 6, we show that every configuration paired with one of its implied stacks has a path leading to it (i.e., the top edge of figure 4).

Lemma 6 (Stacks have paths)**.** *If $(e,\tilde{\rho},\tilde{a}_\kappa) \in \tilde{r}$ such that $wf(\tilde{\xi},(\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa))$, then:*

$$\forall \tilde{\psi} \in_\psi \tilde{a}_\kappa \text{ (via } \tilde{\sigma}_\kappa).\ ((e_0,\varnothing,\tilde{a}_{halt}),\epsilon) \tilde{\hookrightarrow} ((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \text{ (via } \tilde{\xi},(\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa))$$

Proof. By $wf_{\tilde{r}}(\tilde{\xi},(\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa))$, there is some $\tilde{\psi}' \in_\psi \tilde{a}_\kappa$ (via $\tilde{\sigma}_\kappa$) for which $((e_0,\varnothing,\tilde{a}_{\text{halt}}),\epsilon) \tilde{\hookrightarrow} ((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}')$ (via $\tilde{\xi}$, $(\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa)$). However, this path uses $\tilde{\psi}'$ instead of our desired $\tilde{\psi}$. Thus we induct over $\tilde{\psi}$. If $\tilde{\psi}$ is the empty list, ϵ, then $\tilde{a}_\kappa$ must be $\tilde{a}_{\text{halt}}$ and thus ϵ is the only $\tilde{\psi}$ for which $\tilde{\psi} \in_\psi \tilde{a}_\kappa$ (via $\tilde{\sigma}_\kappa$). So $\tilde{\psi}' = \tilde{\psi} = \epsilon$, and the path obtained from $wf_{\tilde{r}}(\tilde{\xi},(\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa))$ equals our desired conclusion.

If $\tilde{\psi}$ is $((x, e_\kappa, \tilde{\rho}_\kappa), \tilde{a}'_\kappa) : \tilde{\psi}''$ for some x, e_κ, $\tilde{\rho}_\kappa$, $\tilde{a}'_\kappa$, $\tilde{\psi}''$, then there is a path for $\tilde{\psi}'$ from $\mathit{Entry}(\tilde{a}_\kappa)$ to $(e, \tilde{\rho}, \tilde{a}_\kappa)$. By another induction there is a similar path for $\tilde{\psi}$:

$$(\mathit{Entry}(\tilde{a}_\kappa), \tilde{\psi}) \tilde{\hookrightarrow} ((e, \tilde{\rho}, \tilde{a}_\kappa), \tilde{\psi}) \text{ (via } \tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$$

By $\mathit{wf}_{\tilde{r}}(\tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$, there exist f and æ for a call site $((\texttt{let ([}x\ \texttt{(}f\ \text{æ}\texttt{)])}\ e_\kappa), \tilde{\rho}_\kappa, \tilde{a}'_\kappa) \in \tilde{r}$ and a step from that call site to $\mathit{Entry}(\tilde{a}_\kappa)$:

$$((\texttt{let ([}x\ \texttt{(}f\ \text{æ}\texttt{)])}\ e_\kappa), \tilde{\rho}_\kappa, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}'_\kappa) \overset{\sqsubseteq}{\rightsquigarrow}_{\tilde{\Sigma}} (e', \tilde{\rho}', \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa)$$
$$\text{where} \quad (e', \tilde{\rho}', \tilde{a}_\kappa) = \mathit{Entry}(\tilde{a}_\kappa)$$

By the induction hypothesis, we have a path for $\tilde{\psi}''$ from $(e_0, \varnothing, \tilde{a}_{\text{halt}})$ to the call site:

$$((e_0, \varnothing, \tilde{a}_{\text{halt}}), \epsilon) \tilde{\hookrightarrow} (((\texttt{let ([}x\ \texttt{(}f\ \text{æ}\texttt{)])}\ e_\kappa), \tilde{\rho}_\kappa, \tilde{a}'_\kappa), \tilde{\psi}'')$$

We now have a path from $((e_0, \varnothing, \tilde{a}_{\text{halt}}), \epsilon)$ to the call site with $\tilde{\psi}''$, a step from the call site to $\mathit{Entry}(\tilde{a}_\kappa)$ that pushes $(x, e_\kappa, \tilde{\rho}_\kappa)$ onto the stack, and a path from $\mathit{Entry}(\tilde{a}_\kappa)$ to $(e, \tilde{\rho}, \tilde{a}_\kappa)$ with $\tilde{\psi}$. From these we can then construct the path desired in our conclusion. □

Next, with lemma 7, we show that every path in a well-formed $\tilde{\xi}$ has a corresponding path in any $\hat{\xi}$ that is at at fixed point (i.e., the left edge of figure 4).

Lemma 7 (Path conversion)**.** *If* $(e, \tilde{\rho}, \tilde{a}_\kappa) \in \tilde{r}$ *such that* $\mathit{wf}(\tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$ *and* $(\hat{r}, \hat{\sigma}) \overset{\hat{}}{\rightsquigarrow}_{\hat{\Xi}} (\hat{r}, \hat{\sigma})$*, then:*

$$((e_0, \varnothing, \tilde{a}_{\mathit{halt}}), \epsilon) \tilde{\hookrightarrow} ((e, \tilde{\rho}, \tilde{a}_\kappa), \tilde{\psi}) \text{ (via } \tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$$
$$\implies (e_0, \varnothing, \epsilon) \hat{\hookrightarrow} H_{\mathit{C}}((e, \tilde{\rho}, \tilde{a}_\kappa), \tilde{\psi}) \text{ (via } \hat{r}, \hat{\sigma})$$

Proof. By induction over the finite-state path. We have three cases.

Case: The path is empty. Trivial.

Case: The last step of the path is a return. For some æ, $\tilde{\rho}'$, $\tilde{a}_\kappa$, $\tilde{a}'_\kappa$, and $\tilde{\rho}''$, there is a step $(\text{æ}, \tilde{\rho}', \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}'_\kappa) \overset{\sqsubseteq}{\rightsquigarrow}_{\tilde{\Sigma}} (e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa)$ and, by the induction hypothesis, a path:

$$(e_0, \varnothing, \epsilon) \hat{\hookrightarrow} H_{\mathit{C}}((\text{æ}, \tilde{\rho}', \tilde{a}'_\kappa), ((x, e, \tilde{\rho}''), \tilde{a}_\kappa) : \tilde{\psi}) \text{ (via } \hat{r}, \hat{\sigma})$$
$$\text{where} \quad \tilde{\rho} = \tilde{\rho}''[x \mapsto \widetilde{\mathit{alloc}}(x, (\text{æ}, \tilde{\rho}', \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}'_\kappa))]$$

We can then show that $(\hat{r}, \hat{\sigma})$ contains a step corresponding to the step in $(\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa)$:

$$(\text{æ}, H_{\mathit{Env}}(\tilde{\rho}'), \hat{\sigma}, (x, e, H_{\mathit{Env}}(\tilde{\rho}'')) : H_{\mathit{Kont}}(\tilde{\psi}))$$
$$\overset{\sqsubseteq}{\rightsquigarrow}_{\tilde{\Sigma}} (e, H_{\mathit{Env}}(\tilde{\rho}), \hat{\sigma}, H_{\mathit{Kont}}(\tilde{\psi}))$$

Combining this with the path from the induction hypothesis, we can then construct the path in our conclusion.

Case: The last step of the path is a call. For some x, y, f, æ, e', $\tilde{\rho}'$, $\tilde{\rho}_\lambda$, $\tilde{a}_\kappa$, $\tilde{\psi}'$, we have $(y, e, \tilde{\rho}_\lambda) \in \tilde{\mathcal{A}}(f, \tilde{\rho}', \tilde{\sigma})$, and a step:

$$(\texttt{(let ([}x\ \texttt{(}f\ \text{æ}\texttt{)])}\ e'\texttt{)}, \tilde{\rho}', \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}'_\kappa) \overset{\sqsubseteq}{\rightsquigarrow}_{\tilde{\Sigma}} (e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa), \text{ where}$$
$$\tilde{\rho} = \tilde{\rho}_\lambda[x \mapsto \widetilde{\mathit{alloc}}(x, (\texttt{(let ([}x\ \texttt{(}f\ \text{æ}\texttt{)])}\ e'\texttt{)}, \tilde{\rho}', \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}'_\kappa))]$$

and, by the induction hypothesis, a path:

$$(e_0, \varnothing, \epsilon) \hat{\hookrightarrow} H_{\mathit{C}}((\texttt{(let ([}x\ \texttt{(}f\ \text{æ}\texttt{)])}\ e'\texttt{)}, \tilde{\rho}', \tilde{a}'_\kappa), \tilde{\psi}') \text{ (via } \hat{r}, \hat{\sigma})$$

We can then show that $(\hat{r}, \hat{\sigma})$ contains a step corresponding to the step in $(\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa)$:

$$(\texttt{(let ([}x\ \texttt{(}f\ \text{æ}\texttt{)])}\ e'\texttt{)}, H_{\mathit{Env}}(\tilde{\rho}'), \hat{\sigma}, H_{\mathit{Kont}}(\tilde{\psi}'))$$
$$\overset{\sqsubseteq}{\rightsquigarrow}_{\tilde{\Sigma}} (e, H_{\mathit{Env}}(\tilde{\rho}), \hat{\sigma}, (x, e', H_{\mathit{Env}}(\tilde{\rho}')) : H_{\mathit{Kont}}(\tilde{\psi}'))$$

Combining this with the path from the induction hypothesis, we can then construct the path in our conclusion. □

Then, with lemma 8, we show that the endpoint of any path in $\hat{\xi}$ is in $\hat{\xi}$ (i.e., the bottom edge of figure 4).

Lemma 8 (Path endpoint)**.** *If* $(\hat{r}, \hat{\sigma}) \overset{\hat{}}{\rightsquigarrow}_{\hat{\Sigma}} (\hat{r}, \hat{\sigma})$*, then for any path* $\hat{c}_0 \hat{\hookrightarrow} (e, \hat{\rho}, \hat{\kappa})$ *(via* $\hat{r}$*,* $\hat{\sigma}$*), we have:* $(e, \hat{\rho}, \hat{\kappa}) \in \hat{r}$*.*

Proof. Trivial. By induction. □

Finally, with lemmas 9 and 10, we show that precision is preserved by the step relation $\tilde{\rightsquigarrow}_{\tilde{\Sigma}}$ (i.e., the right edge of figure 4). Then in theorem 11, we show that these are all precise, which is ultimately what we want to prove.

Lemma 9 (Preservation of precision for value stores)**.** *If* $(\hat{r}, \hat{\sigma}) \overset{\hat{}}{\rightsquigarrow}_{\hat{\Xi}} (\hat{r}, \hat{\sigma})$*,* $\mathit{wf}(\tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$*,* $(\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa) \tilde{\rightsquigarrow}_{\tilde{\Xi}} (\tilde{r}', \tilde{\sigma}', \tilde{\sigma}'_\kappa)$*, and* $(\hat{r}, \hat{\sigma}) \sqsupseteq_{\Xi} (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa)$*, then* $\hat{\sigma} \sqsupseteq_{\mathit{Store}} \tilde{\sigma}'$*.*

Proof. Omitted for space. □

Lemma 10 (Preservation of precision for reachable configurations)**.** *If* $(\hat{r}, \hat{\sigma}) \overset{\hat{}}{\rightsquigarrow}_{\hat{\Xi}} (\hat{r}, \hat{\sigma})$*,* $\mathit{wf}(\tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$*,* $(\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa) \tilde{\rightsquigarrow}_{\tilde{\Xi}} (\tilde{r}', \tilde{\sigma}', \tilde{\sigma}'_\kappa)$*, and* $(\hat{r}, \hat{\sigma}) \sqsupseteq_{\Xi} (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa)$*, then* $\hat{r} \sqsupseteq_{R} \tilde{r}'$*.*

Proof. If we unfold the definition of $(\sqsupseteq_R)$, we must show that for all $(e, \hat{\rho}, \hat{\kappa}) \in \hat{r}'$ and $\tilde{\psi} \in_\psi \tilde{a}_\kappa$ (via $\tilde{\sigma}'_\kappa$), that $(e, H_{\mathit{Env}}(\tilde{\rho}), H_{\mathit{Kont}}(\tilde{\psi})) \in \hat{r}$. By lemma 6, we have:

$$((e_0, \varnothing, \tilde{a}_{\text{halt}}), \epsilon) \tilde{\hookrightarrow} ((e, \tilde{\rho}, \tilde{a}_\kappa), \tilde{\psi}) \text{ (via } \tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$$

From this, by lemma 7, we have:

$$(e_0, \varnothing, \epsilon) \hat{\hookrightarrow} (e, H_{\mathit{Env}}(\tilde{\rho}), H_{\mathit{Kont}}(\tilde{\psi})) \text{ (via } \hat{r}, \hat{\sigma})$$

Finally, by lemma 8, we have our conclusion. □

Theorem 11 (Precision of analysis results)**.** *If* $\tilde{\xi}$ *is the result of taking zero or more steps of* $(\tilde{\rightsquigarrow}_{\tilde{\Xi}})$*, starting from* $(\{(e_0, \varnothing, \tilde{a}_{\mathit{halt}})\}, \bot, \bot)$*, and* $\hat{\xi} \overset{\hat{}}{\rightsquigarrow}_{\hat{\Xi}} \hat{\xi}$*, then* $\hat{\xi} \sqsupseteq_{\Xi} \tilde{\xi}$*.*

Proof. By induction over the number of steps, trivial simplifications, unfoldings, and lemmas 5, 9, and 10. □

6. Conclusion

Traditional control-flow analysis has long suffered from return-flow conflation of values, even when context sensitivity and related techniques keep these values separate across function calls. Recent approaches have made significant progress in addressing this problem. However, each suffers from serious drawbacks. PDCFA incurs a substantial development cost and causes a quadratic-factor increase in run-time complexity. AAC is trivial to implement, but incurs an even worse increase in run-time complexity. Our approach, however, both is simple to implement and adds no asymptotic cost to run-time complexity. To accomplish this, we synthesize the lessons learned from PDCFA and AAC to show that the ideal continuation address is simply a function's polyvariant entry point: its expression and abstract binding environment. This introspection on entry points and the corresponding choice of continuation address yields a finite-state analysis whose call transitions are precisely matched with return transitions at no cost to either run-time or development-time overhead.

Acknowledgments

This material is partially based on research sponsored by DARPA under agreement numbers AFRL FA8750-15-2-0092 and FA8750-12-2-0106 and by NSF under CAREER grant 1350344. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Paris, France, 1976.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, Jan. 1977. ACM. doi: 10.1145/512950.512973.

[3] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282, New York, NY, USA, Jan. 1979. ACM. doi: 10.1145/567752.567778.

[4] C. Earl, M. Might, and D. Van Horn. Pushdown control-flow analysis of higher-order programs: Precise, polyvariant and polynomial-time. In *Scheme Workshop*, August 2010.

[5] C. Earl, I. Sergey, M. Might, and D. Van Horn. Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 177–188, New York, NY, USA, Sept. 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364576.

[6] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, Aug. 1993. ACM. ISBN 0-89791-598-4. doi: 10.1145/155090.155113.

[7] J. I. Johnson. AAC complexity analysis discussion. Unpublished correspondence, June 2015.

[8] J. I. Johnson and D. Van Horn. Abstracting abstract control. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 11–22, New York, NY, USA, Oct. 2014. ACM. ISBN 978-1-4503-3211-8. doi: 10.1145/2661088.2661098.

[9] J. I. Johnson, N. Labich, M. Might, and D. Van Horn. Optimizing abstract abstract machines. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 443–454, New York, NY, USA, Sept. 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500604.

[10] J. Midtgaard. Control-flow analysis of functional programs. *ACM Computing Surveys (CSUR)*, 44(3):10:1–10:33, June 2012. ISSN 0360-0300. doi: 10.1145/2187671.2187672.

[11] M. Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 2007.

[12] M. Might. Abstract interpreters for free. In R. Cousot and M. Martel, editors, *Proceedings of the Static Analysis Symposium*, volume 6337 of *Lecture Notes in Computer Science*, pages 407–421. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15768-4. doi: 10.1007/978-3-642-15769-1_25.

[13] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 305–315, New York, NY, USA, June 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806631.

[14] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[15] D. Van Horn and M. Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 51–62, New York, NY, USA, Sept. 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863553.

[16] D. Vardoulakis and O. Shivers. CFA2: A context-free approach to control-flow analysis. In A. D. Gordon, editor, *Proceedings of the European Symposium on Programming*, volume 6012 of *Lecture Notes in Computer Science*, pages 570–589. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-11956-9. doi: 10.1007/978-3-642-11957-6_30.