Pushdown Control-Flow Analysis for Free

Thomas Gilray Steven Lyde Michael D. Adams Matthew Might David Van Horn
University of Utah University of Maryland
{tgilray,lyde,adamsmd,might}@cs.utah.edu, dvanhorn@cs.umd.edu

Abstract

Traditional control-flow analysis (CFA) for higher-order languages, whether implemented by constraint-solving or abstract interpretation, introduces spurious connections between callers and callees. Two distinct invocations of a function will necessarily pollute one another's return-flow. Recently, three distinct approaches have been published which provide perfect call-stack precision in a computable manner: CFA2, PDCFA, and AAC. Unfortunately, CFA2 and PDCFA are difficult to implement and require 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 $O(n^9 \log n)$.

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 continuation. Our technique imposes only a constant-factor overhead on the underlying analysis and, with monovariance, costs only $O(n^3)$ in the worst case.

This paper presents the intuitions behind this development, a proof of the precision of this analysis, and benchmarks demonstrating its efficacy.

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

Keywords static analysis, abstract interpretation, verification, pushdown, continuation, allocation, 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 code:

```
(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`.

To avoid this imprecision, Vardoulakis and Shivers introduced their *context-free approach* (as in context-free languages, not context-sensitivity) to program analysis with CFA2 [13]. This technique provides a computable, although exponential-time, method for obtaining perfect stack-precision for monovariant analyses of continuation-passing-style programs. Two different approaches, PDCFA and AAC, build on this work by enabling polyvariant (e.g. context-sensitive) analysis of direct-style programs and at only a polynomial-factor increase to the run-time complexity of the underlying analysis.

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

Johnson and Van Horn's *abstracting abstract control* (AAC) uses a refinement of store-allocated continuations with the established finite-state method of merging stack frames into the store, and they define an allocator precise enough to avoid all spurious merging [7]. The key advantage of this method is that it is trivial to implement in existing analysis frameworks that use store-allocated continuations, coming at the cost of changing roughly one line of code. Unfortunately, AAC is significantly more complex than PDCFA; even in the monovariant case it is $O(n^9 \log n)$.

We draw on the lessons learned from all three approaches and present a technique obtaining the same perfect call-stack precision at only a constant-factor increase to run-time complexity over finite-state analysis (*for free* in terms of complexity) and requiring no refactoring of analyses already using store-allocated continuations (*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 the store-allocated continuations approach. In the monovariant case, our analysis remains 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 which demonstrate its efficacy.
- We define a relationship between our technique and a static analysis using unabstracted (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, the relevant background on abstract interpretation using abstract-machines, soundness, 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 which 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 the existing polynomial-time approaches to obtaining an equivalent stack-precision: PDCFA and AAC.

In section 4, we formalize our technique, give the intuitions which lead us to it, and explain how it relates to each of the analyses described in section 3. We describe our implementation and present benchmarks, using both monovariant and call-sensitive allocation, 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 finite-state analysis. We use this relationship to prove the maximal precision of the method we introduce in this paper.

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 analysis of programming languages serving applications such as program verification, malware/vulnerability detection, compiler optimization, among others [1–3, 9]. The *abstracting abstract machines* (AAM) approach uses abstract interpretation for *control-flow analysis* (CFA) of functional (higher-order) programming languages [8, 11, 12]. 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 will present a concrete interpretation of a simple direct-style language, a traditional finite-state abstraction, a store-widened polynomial-time analysis, and explore the return-flow merging problem in greater detail.

2.1 Concrete semantics

We will be using a direct-style (call-by-value, type-free) λ-calculus in administrative normal-form (ANF):

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

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

Our concrete interpreter operates over machine states ς:

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

Binding-environments (environments) ρ, map variables in scope to a visible representative address a. Value-stores (stores) σ, map these addresses to a program value (for pure λ-calculus all values are closures). 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. Note that such a ϕ contains no more or less information than a closure. 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 as there are two simple but distinct cases which are easier to break-out and encapsulate on their own:

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

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 will stop when the end of a program's execution is reached, or when given an invalid state. Call-sites transition according to the first of two transition rules:

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

A new frame ϕ is pushed onto the stack for eventually returning inside 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 which may be different; these do not have a 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 ae to produce σ'.

Return-points transition according to a second rule:

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

The top stack frame ϕ is opened 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 ae to produce σ'. The expression e in the top stack frame is reinstated at ρ' and σ' 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 $\mathcal{I} : \mathsf{Exp} \to \Sigma$ defined:

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

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

$$
\begin{array}{c}
s \in S = \mathcal{P}(\Sigma) \\
s \rightsquigarrow_s s' \iff s' = \{\varsigma' \mid \varsigma \in s \wedge \varsigma \rightsquigarrow_\Sigma \varsigma'\} \cup \{\mathcal{I}(e_0)\}
\end{array}
$$

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

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

2.2 Abstract semantics

We are now ready to design a computable approximation of this fix-point (the exact program trace) using an abstract semantics. Previous work has explored a wide variety of approaches to systematically abstracting a semantics like these [8, 11, 12]. 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 need 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:

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

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 then represented by an address. This address indicates a *set* of top frames, each paired with the address of its continuation in turn (that stack's tail). We factor the continuation-store $\tilde{\sigma}_\kappa$ and value-store $\tilde{\sigma}$ at the top-level to maintain simplicity as we progress.

Abstract environments $\tilde{\rho}$ have changed only because our address set is now finite; abstract closures $\widetilde{clo}$ are now 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 $\tilde{\sigma}_\kappa$ indicates a set of continuations $\tilde{k}$. Like closures, each abstract frame $\tilde{\phi}$ is only approximate 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 $\tilde{\mathcal{A}}$:

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

$$\tilde{\mathcal{A}}(x, \tilde{\rho}, \tilde{\sigma}) = \tilde{\sigma}(\tilde{\rho}(x)) \qquad \text{[variable lookup]}$$

$$\tilde{\mathcal{A}}(lam, \tilde{\rho}, \tilde{\sigma}) = \{(lam, \tilde{\rho})\} \qquad \text{[closure creation]}$$

Note that closure creation now yields a singleton-set.

Because our address domain is now finite, multiple concrete allocations will need to be represented by a single abstract address. There are a variety of sound strategies for doing this; each 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 to allocate for, the finite-state performing the allocation, and the closures invoked, 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:

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

This tunes our finite-state semantics to the *monovariant* analysis style, a form of context-insensitive analysis. In a monovariant analysis, every closure which is bound to a variable x, at any point during a concrete execution, will end up being represented in a single flow-set when the analysis is complete. A more precise example of an allocation behavior will be given later in this section.

Because we are also now store-allocating continuations and distinguishing a top-level continuation-store, we likewise distingush 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:

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

We may provide this function all the information known about the transition being made. The standard allocator is invoked before a successor $\tilde{\rho}'$ or $\tilde{\sigma}'$ is constructed; however, when calling the continuation-allocator, we may provide information about the target state being transitioned to. The choice of e' for allocating a continuation-address makes a lot of sense considering the entry-point of a function should in a sense *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 alphatization where each x is unique to a single binding-point).

We may now define a non-deterministic finite-state transition relation $(\rightsquigarrow_{\tilde{\Sigma}}) \in \tilde{\Sigma} \times \tilde{\Sigma}$. Call-sites transition according to a rule:

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

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

As $\tilde{\mathcal{A}}$ yields a set of abstract closures for f, a successor state is produced for each across this transition. Likewise, for each point in the store to accumulate all closures bound at that abstract address $\tilde{a}$

and to represent a faithful over-approximation of all the addresses a which $\tilde{a}$ simulates, we use a join operation when extending the store. The join of two stores distributes point-wise, that is:

$$(\tilde{\sigma}_1 \sqcup \tilde{\sigma}_2)(\tilde{a}) = \tilde{\sigma}_1(\tilde{a}) \cup \tilde{\sigma}_2(\tilde{a})$$

Instead of generating a fresh address for $\tilde{a}$, we use our abstract allocation policy to generate 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.

A return transition is as before except modified in the same way:

$$\overbrace{(\text{æ}, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa)}^{\tilde{\varsigma}} \rightsquigarrow_\Sigma^\sim (e, \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}_\kappa, \tilde{a}'_\kappa), \text{ where}$$
$$((x, e, \tilde{\rho}_\kappa), \tilde{a}'_\kappa) \in \tilde{\sigma}_\kappa(\tilde{a}_\kappa)$$
$$\tilde{\rho}' = \tilde{\rho}_\kappa[x \mapsto \tilde{a}]$$
$$\tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a} \mapsto \tilde{\mathcal{A}}(\text{æ}, \tilde{\rho}, \tilde{\sigma})]$$
$$\tilde{a} = \widetilde{alloc}(x, \tilde{\varsigma})$$

Where multiple top stack frames are indicated by $\tilde{a}_\kappa$, this transition yields multiple successors (one for each). An updated environment and store are produced as before, however the continuation-store remains as it was. The current continuation $\tilde{a}'_\kappa$ reinstated in each successor is the address associated with each top stack frame.

To approximately evaluate a program according to these abstract semantics, we first define an abstract injection function:

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

Where the store (now a total function) begins as a function indicating the empty set for each abstract address.

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

The address a_{halt} can be any otherwise unused value that will never be returned by the allocation function. Our machine will eventually be unable to transition into this continuation and will then produce no successors, simulating the behavior of our concrete machine upon reaching an empty stack ϵ.

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

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

Our collecting relation $(\rightsquigarrow_{\tilde{s}}^\sim)$ is a monotone total-function which 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 $(\rightsquigarrow_{\tilde{s}}^\sim)^n(\bot)$ is guarenteed to be a fix-point containing an approximation of e_0's full program trace.

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—we are placing an upper bound on the propagation of closures though a program.

To establish such a relationship between a concrete and abstract semantics, we use a Galois-connection. A Galois-connection is a pair of functions for abstraction and concretization:

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

such that

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

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:

$$\alpha(s) \subseteq \tilde{s} \wedge s \rightsquigarrow_s s' \implies \tilde{s} \rightsquigarrow_{\tilde{s}}^\sim \tilde{s}' \wedge \alpha(s') \subseteq \tilde{s}'$$

Diagrammatically:

$$\begin{array}{ccc} s & \xrightarrow{\rightsquigarrow_s} & s' \\ \subseteq \downarrow \alpha & & \subseteq \downarrow \alpha \\ \tilde{s} & \xrightarrow{\rightsquigarrow_{\tilde{s}}^\sim} & \tilde{s}' \end{array}$$

Both constructing analyses using Galois-connections and proving them sound using Galois-connections has been extensively explored in the literature [11**?** , 12]. The analysis style we constructed in section 2.2 has been previously proven sound using the above method [10].

2.4 Store-widening

Various forms of widening and further approximation may be layered on top of what we have already done. One such form is essential and required for an analysis to be tractable (polynomial-time): store-widening. To see why store-widening is necessary, let us consider the complexity of an analysis using $(\rightsquigarrow_{\tilde{s}}^\sim)$. 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. For the imprecise allocators we've defined, analysis run-time is in:

$$O(\overbrace{n}^{|\mathsf{Exp}|} \times \overbrace{n}^{|\widetilde{Env}|} \times \overbrace{2^{n^2}}^{|\widetilde{Store}|} \times \overbrace{2^{n^3}}^{|\widetilde{KStore}|} \times \overbrace{n}^{|\widetilde{Addr}|})$$

The number of syntactic points in an input program is in $O(n)$. The number of environments may be a large polynomial, or even an exponential, given a more precise choice for addresses produced by $\widetilde{alloc}$; in the monovariant case however, 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 stores may be visualized as a table of possible mappings from every address to every abstract closure—each of these may be included in a given store or not. The number of abstract closures is in $O(n)$ (lambdas uniquely determine a monovariant environment), times the number of addresses gives $O(n^2)$ possible additions to the value-store. The number of abstract frames is in $O(n)$ (let-forms uniquely determine their binding variable, body, and monovariant environment), times the number of possible continuation addresses gives $O(n^2)$, and times the number of addresses gives $O(n^3)$.

The crux of the issue we're illustrating is that, in exploring states non-deterministically where each state is specific to a whole store, we may explore both sides of every diamond in the store lattice. All combinations of possible bindings in the 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 ($\widetilde{clo}_2$ before $\widetilde{clo}_1$). We might also extend another address $\tilde{a}_2$ with $\widetilde{clo}_1$ either before or after either of these case, and so forth.

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 $\tilde{S}$. The analysis then strictly accumulates new values in the store in the same way it accumulates

reachable states instead of accumulating whole stores and therefore all possible combinations of additions within these stores. To formalize this, we define new *widened* state-spaces which pair a set of reachable configurations with a global value-store and global continuation-store:

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

A widened transfer function may then be defined which, like $(\rightsquigarrow_{\tilde{s}})$, is a total function we may iterate to a fix-point.

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

This may be defined in terms of $(\rightsquigarrow_{\tilde{\Sigma}})$ 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:

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

An underscore (wildcard) matches anything without binding. The height of the $\tilde{\Xi}$ lattice is now cubic, more specifically it is in:

$$O(\overbrace{n}^{|\mathsf{Exp}|} \times \overbrace{n}^{|\widetilde{Env}|} \times \overbrace{n}^{|\widetilde{Addr}|} + \overbrace{n^2}^{|\widetilde{Store}|} + \overbrace{n^3}^{|\widetilde{KStore}|})$$

2.5 Stack-imprecision

To more easily illustrate the effect of an imprecise stack on data-flow and control-flow precision, we first define a more precise 1-call-sensitive (i.e. 1-CFA) allocator. A k-call-sensitive analysis style will differentiate bindings to a variable so they are additionally unique to a history of the last k call-sites reached before the binding. A history of length 1 would then allocate an address unique to the call-site immediately preceeding the binding:

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

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

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

We number these expressions for ease of reference; e.g. e_2 refers to the let-form binding `z`, e_0 to the return-point of `id`.

We assume the starting configuration $\tilde{c}$ for this example is $(e_1, \tilde{\rho}, \tilde{a}_\kappa)$, so $\tilde{\rho}$ and $\tilde{a}_\kappa$ are the binding environment and continuation at the start of this code.

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

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

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

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

Next, `id` returns and transitions from e_0 to e_3, reinstating the continuation address $\tilde{a}_\kappa$. Because e_0 is bound to both continuations, this transition binds (z, e_0) to `#t` and also (y, e_0) to `#t`, causing return-flow imprecision and giving us the following stores:

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

The address $(\mathtt{y}, e_0)$ representing `y` in e_3 contains both `#t` and `#f`, even though no concrete execution binds `y` to `#f`. Another transition will cause the same conflation for `z`.

Clearly one solution is to increase the context-sensitivity of our continuation allocator. Consider a continuation allocator which, like $\widetilde{alloc}_1$, uses a single call-site of context and allocates an address (e', e) for continuations formed from both e' the expression being transitioned to, and e the expresson being transitioned from. This results in no spurious merging at return-points because continuations are kept as distinct as the value-store addresses we allocate.

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, albiet at significant engineering and run-time costs, can perfect stack-precision be implemented as a *fixed* continuation allocator? In this paper we will be able to both answer this question in the affirmative and show that this leads us not only to a trivial implementation, but to a constant-factor increase to run-time complexity.

3. Perfect Stack-Precision

We next formalize what is meant by a static analysis with *perfect stack-precision* using an abstract abstract-machine with unbounded stacks within each machine state. 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

We formalize this kind of perfect precision in the same manner as previous work on this topic: using a static analysis which 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. Environments, value-stores, and closures are otherwise abstracted in the same manner as the finite machine of section 2.2. Components unique to this machine wear hats:

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

Our atomic-expression evaluator works just as before:

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

As would a monovariant allocator:

$$\widehat{alloc} : \mathsf{Var} \times \hat{\Sigma} \to \widehat{Addr}$$
$$\widehat{alloc}(x, \hat{\varsigma}) = x$$

This may also be tuned to another allocation strategy as easily as before.

We may now define a non-deterministic pushdown-state transition relation $(\rightsquigarrow_{\hat{\Sigma}}) \in \hat{\Sigma} \times \hat{\Sigma}$ and a rule for call-site transitions:

$$\overbrace{(\texttt{(let ([}x\ (f\ æ)\texttt{]) }e\texttt{)}, \hat{\rho}, \hat{\sigma}, \hat{\kappa})}^{\hat{\varsigma}} \rightsquigarrow_{\hat{\Sigma}} (e', \hat{\rho}', \hat{\sigma}', \hat{\phi} : \hat{\kappa}), \text{ where}$$
$$\hat{\phi} = (x, e, \hat{\rho})$$
$$(\texttt{(}\lambda\ \texttt{(}x\texttt{) }e'\texttt{)}, \hat{\rho}_\lambda) \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$$
$$\hat{\rho}' = \hat{\rho}_\lambda[x \mapsto \hat{a}]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathcal{A}}(æ, \hat{\rho}, \hat{\sigma})]$$
$$\hat{a} = \widehat{alloc}(x, \hat{\varsigma})$$

This is familiar because it has been simplified slightly 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 done away. Instead, this transition yields an unbounded stack $\hat{\phi} : \hat{\kappa}$ for each successor with a new top-frame $\hat{\phi}$ containing the variable to bind x, let-body e, and environment to reinstate $\hat{\rho}$.

Likewise, a return transition is straightforward:

$$\overbrace{(æ, \hat{\rho}, \hat{\sigma}, \hat{\phi} : \hat{\kappa})}^{\hat{\varsigma}} \rightsquigarrow_{\hat{\Sigma}} (e, \hat{\rho}', \hat{\sigma}', \hat{\kappa}), \text{ where}$$
$$\hat{\phi} = (x, e, \hat{\rho}_\kappa)$$
$$\hat{\rho}' = \hat{\rho}_\kappa[x \mapsto \hat{a}]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathcal{A}}(æ, \hat{\rho}, \hat{\sigma})]$$
$$\hat{a} = \widehat{alloc}(x, \hat{\varsigma})$$

To return, the stack must contain at least one frame. The indicated e is reinstated within the environment $\hat{\rho}$ extended with an address for x. The store is extended and whatever stack-tail existed under $\hat{\phi}$ (empty or not) is the successor's current-continuation $\hat{\kappa}$.

Pushdown-state injection is defined as we'd expect:

$$\hat{\mathcal{I}} : \mathsf{Exp} \to \hat{\Sigma}$$
$$\hat{\mathcal{I}}(e) = (e, \varnothing, \bot, \epsilon)$$

As before, we lift $(\rightsquigarrow_{\hat{\Sigma}})$ to obtain a monotonic naïve state-space transfer function $(\rightsquigarrow_{\hat{s}})$ defined over sets of states:

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

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

3.2 Global store-widened $\hat{\Sigma}$-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:

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

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:

$$(\rightsquigarrow_{\hat{\Xi}}) : \hat{\Xi} \to \hat{\Xi}$$
$$(\hat{r}, \hat{\sigma}) \rightsquigarrow_{\hat{\Xi}} (\hat{r}', \hat{\sigma}'), \text{ where}$$
$$\hat{s} = \{\hat{\varsigma} : (e, \hat{\rho}, \hat{\kappa}) \in \hat{r} \wedge (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \rightsquigarrow_{\hat{\Sigma}} \hat{\varsigma}\} \cup \{\hat{\mathcal{I}}(e_0)\}$$
$$\hat{r}' = \{(e, \hat{\rho}, \hat{\kappa}) : (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \in \hat{s}\}$$
$$\hat{\sigma}' = \bigsqcup_{(\_,\_,\hat{\sigma},\_) \in \hat{s}} \hat{\sigma}$$

3.3 Pushdown control-flow analysis (PDCFA)

Pushdown control-flow analysis (PDCFA) is a strategy for creating a computable analysis equivalent to the precision of our unbounded-stack machine at a quadratic-factor increase to the complexity class of the underlying analysis (e.g. monovariant, 1-call-sensitive) [4, 5]. This strategy tracks both reachable states (or in the widened case, configurations) as well as push/pop edges between them. A quadratic blow-up comes from the fact that each pair of reachable states may have an edge between them. These edges implicitly represent, as 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 in an unbounded-stack analysis a sequence of pushes may be repeated ad infinitum, a Dyke state graph equivalent will represent a cycle of pushes and a cycle of pops finitely. Broadly speaking, this is also how AAC and our improved continuation allocator will work, except such cycles are represented in the store.

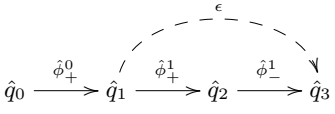
At an intuitive level, a Dyke state graph is a state graph where each transition edge is annotated with either a frame push, a frame pop, or epsilon. The set of continuations for a particular state in a Dyke state graph is determined by the pushes and pops along the paths of the graph that reach that state. In each of these paths a pop of a particular frame cancels out a push of the same frame. In addition, any paths that result a pop after a push of a different frame are ignored.

Pushes and pops along these paths are collected and when corresponding pushes

The PDCFA method requires some extra machinery to be computed extensionally using an iterated transfer function for $\hat{G}$. The primary difficulty is in computing the top stack frame for some configuration $\hat{q}$. To illustrate this, consider the portion of a Dyke state graph:

$$\hat{q}_0 \xrightarrow{\hat{\phi}^0_+} \hat{q}_1 \xrightarrow{\hat{\phi}^1_+} \hat{q}_2 \xrightarrow{\hat{\phi}^1_-} \hat{q}_3$$

The Dyke configuration immediately beneath $\hat{q}_3$ on the stack is not encoded within $\hat{q}_3$ or any of its adjascent nodes. It could return to $\hat{q}_1$, and in the general case, through complex sequences of canceling pushes and pops, possibly an arbitrarily distant configuration. PDCFA therefore requires the inductive maintainence of an *epsilon closure graph* in addition to the Dyke state graph. This structure makes all sequences of canceling stack actions explicit as an epsilon edge:



As we'll see, this epsilon closure graph represents unnecessary additional complexity for both computer and analysis engineer.

We reuse some components of our unbounded-stack machine, continuing to use hats as these machines are so closely related:

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

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

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

Due to space constraints we are unable to fully formalize how a Dyke state graph is computed extensionally (using an monotonic iteration function as we did for the finite and unbounded-stack analyses). Instead, we define it intensionally from a completed unbounded-stack analysis $\hat{\xi}$ and describe some of the key machinery needed to compute it bottom-up. The function $\mathcal{DSG} : \hat{\Xi} \to \hat{G}$ produces a Dyke state graph from a fix-point for $(\hat{\rightsquigarrow}_\Xi)$:

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

$$\hat{v} = \{(e, \hat{\rho}) \mid (e, \hat{\rho}, \hat{\kappa}) \in \hat{r}\}$$

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

3.4 Precise allocation of continuations (AAC)

Abstracting abstract-control (AAC) is another polynomial-time method for obtaining perfect stack-precision [7]. 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 or context sensitivity being used for program values. The method results in a more than cubic-factor increase to run-time complexity, and is given as $O(n^9 \log n)$ in the monovariant/store-widened case by its authors [6]; however, AAC makes perfect stack-precision available *for free* in terms of the engineering cost and manual-time for construction and maintainance of analysis frameworks.

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

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

That is, continuations are stored at an address unique both to the target state's expression and environment and to the source state's expression, environment, and store.

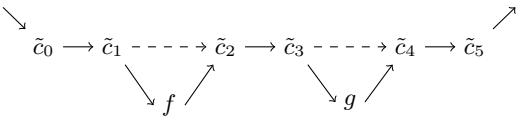
We have necessarily simplified AAC slightly and translated its notation to give this definition in the terms of our framework; a faithful consideration of AAC should point to certain fundamental differences between our framework and theirs. AAC uses an eval-apply semantics which explodes a flow-set into a set of distinct states across every application. The exact address AAC proposes using is $(((\lambda\ (x)\ e'), \tilde{\rho}_\lambda), \widetilde{clo}, \tilde{\sigma})$ (Figure 7 in [7]) where $((\lambda\ (x)\ e'), \tilde{\rho}_\lambda)$ is the target closure of an application, $\widetilde{clo}$ is one particular abstract closure flowing to x, and $\tilde{\sigma}$ is the value-store in the source state. Our components e' and $\tilde{\rho}'$ are isomorphic to the target closure because e' is identical and $\tilde{\rho}'$ is produced from the combination of $\tilde{\rho}_\lambda$ and x. The source state's components e, $\tilde{\rho}$, $\tilde{\sigma}$ are not as specific as $\widetilde{clo}$ and $\tilde{\sigma}$; they do uniquely determine a flow-set $\tilde{d}$ (the result of $\tilde{\mathcal{A}}$ invoked on f) which 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 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 will be encoded in $\tilde{\rho}'$ and inherited by $\widetilde{alloc}^{\text{AAC}}_\kappa$ upon producing an address. Including this 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 or flow-set only adds unnecessary run-time complexity. This optimization extends AAC's core insight to be computationally *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 ($\tilde{c}_0$ 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 $\tilde{c}_0 \to \tilde{c}_1$, is not technically possible in our restricted ANF language, however in the more general case it is. The function 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 optimization is that *all intraprocedural configurations $\tilde{c}_0$ through $\tilde{c}_5$ will neccessarily share the exact same set of genuine continuations*.

We call the set of configurations $\tilde{c}_0$ through $\tilde{c}_5$ an *intraprocedural group* because they are those configurations which 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 which share a single set of continuations. As our finite-state machine represents this set of continuations with a continuation address, if this continuation address is precise enough to uniquely determine an entry-point (e and $\tilde{\rho}$), then it can be used for all configurations in the same intraprocedural group. Thus our allocator may be defined simply:

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

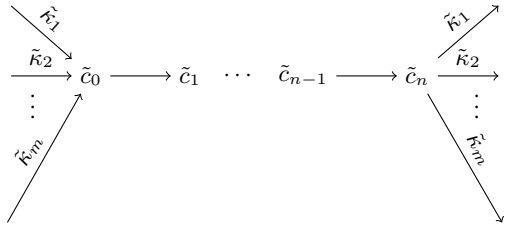
The impact of this change is easily missed, belied by its simplicity. We allocate a continuation based only on the expression and environment at the entry-point of each intraprocedural sequence of let-forms and it is precisely reinstated when each of the calls in these let-forms return.

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 indicated by $\tilde{a}_\kappa$ is less precise than the set of source configurations which transition to the entry-point (e and $\tilde{\rho}$) of the same intraprocedural group. Because we allocate a continuation address specific to this exact entry-point, and because that address is propagated (shallowly copied) to each return-point for the same intraprocedural group, the set of continuations indicated 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 $\widetilde{alloc}_\kappa^{\text{P4F}}$.

We formalize these intuitions and provide a proof that our unbounded-stack analysis simulates a finite-state analysis when using $\widetilde{alloc}_\kappa^{\text{P4F}}$ in section 5.

4.1 Complexity

In order to see, why this allocation scheme leads to a constant-factor overhead, consider a set of configurations, $\tilde{c}_0, \tilde{c}_1, \cdots, \tilde{c}_n$, forming an intraprocedural group, with a set of different call sites transitioning to $\tilde{c}_0$ with the continuations $\tilde{\kappa}_1, \tilde{\kappa}_2, \cdots, \tilde{\kappa}_m$. 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 continuation store then maps this address to the set $\{\tilde{\kappa}_1, \tilde{\kappa}_2, \cdots, \tilde{\kappa}_m\}$.

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 transition, say $\tilde{\kappa}_{m+1}$, in the set at the continuation address $(e, \tilde{\rho})$. Then the edge transitions for the return edge must be added. Note that none of $\tilde{c}_0, \tilde{c}_1, \cdots, \tilde{c}_n$ need to be modified. The only work done here beyond what the underlying analysis does is adding $\tilde{\kappa}_{m+1}$ to the set in the continuation store. Thus, the additional work done by this is a constant factor of number of times a continuation is added to the continuation store.

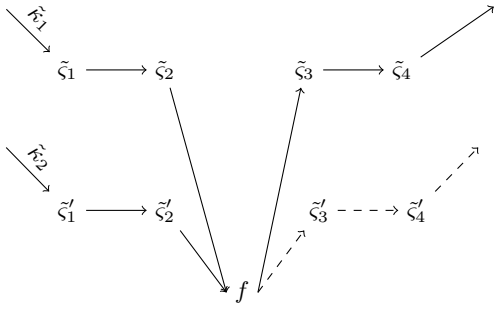
A naive analysis might lead us to conclude that this is bounded by the product of the number of continuation addresses and the number of continuations. However, there is a tighter bound. Each continuation is unique to one specific transition, and that transition stores its continuation at only one specific continuation address, $(e, \tilde{\rho})$, in the continuation store. Thus the work done is a constant factor of the number of continuations, which in turn is bounded by 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: one for each combination of components e, $\tilde{\rho}$, and $\tilde{\sigma}$ drawn from the source-states transitioning to it. This also differs from PDCFA which is not able to shallowly copy a set of incoming epsilon edges but must recompute this set for each intermediate configuration $\tilde{c}_0, \tilde{c}_1, \cdots, \tilde{c}_n$; the number of epsilon edges it may need to compute is therefore in $O(m * n)$, causing a quadratic-factor increase in complexity.

4.2 Restrictions and relations to other systems

That no function can have two entry-points which lead to the same exit-point is a genuine restriction worth discussing further; if it were not true, our technique would be precise (assuming multiple-entry points are not merged), but it would not be only 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 without adding significant complexity to the concrete semantics; for this reason store-widening is required both to obtain a polynomial-time analysis in the first place, and to ensure our technique only results in a constant-factor overhead on top of this.

To see how this can happen, consider a function that is called with two different continuations and two different stores. Without store widening, each store causes a different states to be created for the entry point of the function. For example in the following diagram, $\tilde{\varsigma}_1$ might be the states for the entry point with one store and $\tilde{\varsigma}'_1$ the states for the entry point with the other store.



Now suppose both make a call to some function f, and that f contains some side effect that causes the stores to become equal. For example, perhaps in one store a particular variable, say x, maps to $\{\texttt{\#t}\}$ and in the other store it maps to $\{\texttt{\#f}\}$, and then some side effect in f causes x to map to $\{\texttt{\#t}, \texttt{\#f}\}$ in both stores.

The problem that arises is whether f should return to only one states such as $\tilde{\varsigma}_3$ for both calls or if it should return to two states such as both $\tilde{\varsigma}_3$ and $\tilde{\varsigma}'_3$. Either choice has drawbacks.

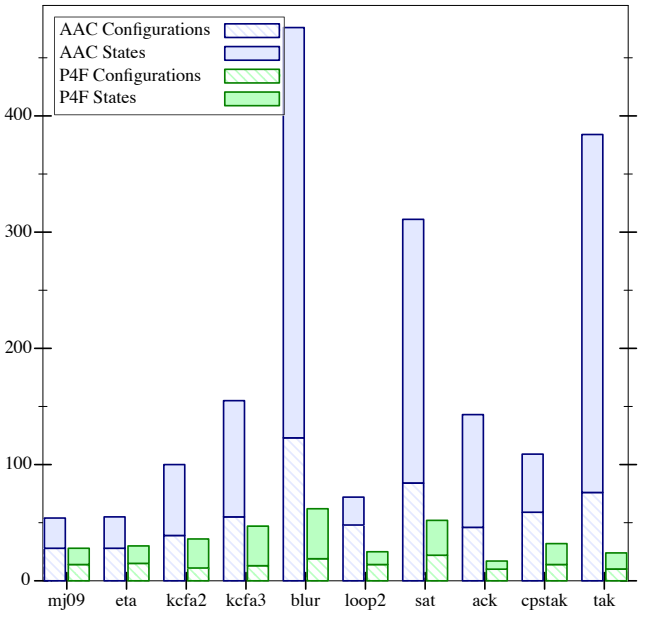


Figure 1. A monovariant comparison.

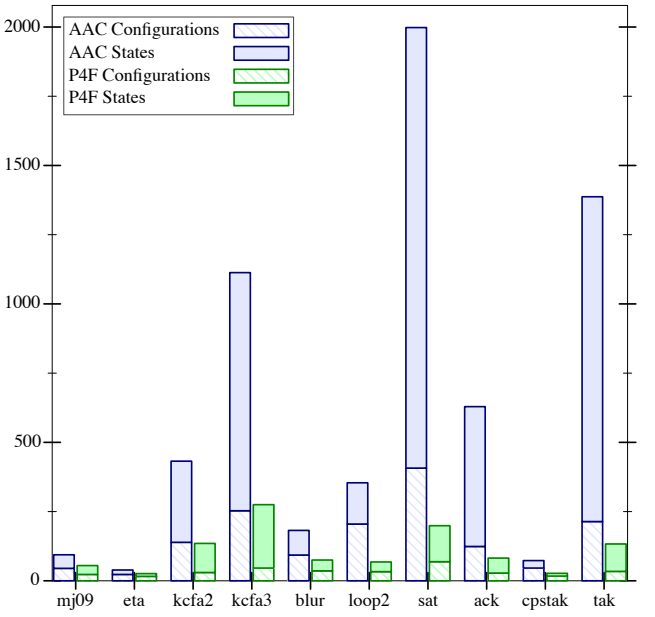


Figure 2. A 1-call-sensitive comparison.

Since after the call to f the stores are the same, one could argue that their should be only one states after the call to f. However, that means that continuation address in $\tilde{\varsigma}_3$ must point to both $\tilde{\kappa}_1$ and $\tilde{\kappa}_2$, even though the continuation address in $\tilde{\varsigma}_2$ points to only $\tilde{\kappa}_1$. This prevents $\tilde{\varsigma}_2$ and $\tilde{\varsigma}_3$ from using the same continuation address and is incompatible with our technique.

On the other hand, we could keep the states separate so we have both $\tilde{\varsigma}_3$ and $\tilde{\varsigma}'_3$. Since each state has its own continuation address, $\tilde{\varsigma}_3$ points to only $\tilde{\kappa}_1$ and $\tilde{\varsigma}'_3$ points to only $\tilde{\kappa}_2$. This means that $\tilde{\varsigma}_1$, $\tilde{\varsigma}_2$, $\tilde{\varsigma}_3$, and $\tilde{\varsigma}_4$ all point to the same continuation as each other and can share the same continuation address. Likewise for $\tilde{\varsigma}'_1$, $\tilde{\varsigma}'_2$, $\tilde{\varsigma}'_3$, and $\tilde{\varsigma}'_4$. While this is compatible with our technique (and indeed is what our technique would naturally do in such a situation), doing this loses the guarantee of costing only a constant factor overhead. This is because the continuations are the only things different between each $\tilde{\varsigma}_i$ and $\tilde{\varsigma}'_i$. The underlying analysis does not cause this distinction. Instead the precision of our analysis forces to pay for the cost of duplicating these states.

It is straightforward to compare the complexity of AAC to our method because it 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 they will produce two different configurations $\tilde{c}'_0, \tilde{c}'_1$ because the continuation-address they allocate will be different. This differentiation will be maintained through two variants of the function starting at e', and when an exit-point æ is reached for each, this expression and its environment will be the same and propagate the same values to (artificially) distinct sets of continuations. These continuation-addresses and their stacks were 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 top 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 maintains a separate set of incoming epsilon edges for each and every such node. This means that the number of such edges for any given $(e, \hat{\rho})$ entry-point is the number of callers times the number of intraprocedural nodes: 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 $(e, \hat{\rho})$ entry-point shares the same set of continuations (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 constant-factor-PDCFA, however not without added machinery which complicates design and maintainence of analyses above and beyond the complexity added by the Dyke state graph approach to begin with.

4.3 Implementation

We have implemented both our technique and AAC's technique for analysis of a simple 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. which implements both traditional k-CFA and PDCFA [5]. The test cases we ran came from the larceny R6RS benchmark suite (ack, cpstak, tak) and examples compiled from the previous literature on obtaining perfect stack-precision (mj09, eta, kcfa2, kcfa3, blur, loop2, sat—these were also used to benchmark PDCFA [5]). 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 1), and a 1-call-sensitive polyvariant allocator (Figure 2). Across the board our method requires visiting strictly fewer machine configurations. In some of these cases the difference is rather small; in others it is significant—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. These numbers would correspond roughly to an average of five distinct callers for each function entry-point. That this number decreased slightly in the context-sensitive case may be attributed to an increase in precision overall.

5. Proof of Precision

To illustrate more formally how our optimization yields a finite-state abstraction with equivalent precision to PDCFA and AAC, we define a simulation relation ($\widehat{\sqsupseteq}_\Xi$) where $\hat{\xi} \widehat{\sqsupseteq}_\Xi \tilde{\xi}$ (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 $\hat{\xi}$. The reader should also recall our discussion of soundness in section 2.3 as this represents the other half ($\sqsubseteq$) of what is more fully a bidirectional isomorphism between these machines. In this section we focus exclusively on the portion of this isomorphism that is novel in our work: precision. We begin by defining the simulation relation between the corresponding parts of the machines:

$$
\begin{aligned}
(\widehat{\sqsupseteq}_\Xi) &\subseteq \hat{\Xi} \times \tilde{\Xi} && \text{[state-space simulation]}\\
(\widehat{\sqsupseteq}_\mathcal{X}) &\subseteq \hat{\Xi} \times \tilde{\Sigma} && \text{[partition simulation]}\\
(\widehat{\sqsupseteq}_{Env}) &\subseteq \widehat{Env} \times \widetilde{Env} && \text{[env. simulation]}\\
(\widehat{\sqsupseteq}_{Store}) &\subseteq \widehat{Store} \times \widetilde{Store} && \text{[store simulation]}\\
(\widehat{\sqsupseteq}_D) &\subseteq \hat{D} \times \tilde{D} && \text{[flow-set simulation]}\\
(\widehat{\sqsupseteq}_{Clo}) &\subseteq \widehat{Clo} \times \widetilde{Clo} && \text{[closure simulation]}\\
(\widehat{\sqsupseteq}_{Kont}) &\subseteq \hat{K} \times \widehat{Addr} \times \widetilde{KStore} && \text{[kont simulation]}
\end{aligned}
$$

We will use $\hat{k} \in \hat{K} = \mathcal{P}(\widehat{Kont})$ to denote sets of unbounded-stacks. This domain simulates $\tilde{K}$ in the finite-state machine. Our ternary ($\widehat{\sqsupseteq}_{Kont}$) relation will be styled like so:

$$(\hat{k}, \tilde{a}_\kappa, \tilde{\sigma}_\kappa) \in (\widehat{\sqsupseteq}_{Kont}) \iff \hat{k} \widehat{\sqsupseteq}_R \tilde{a}_\kappa \text{ (via } \tilde{\sigma}_\kappa)$$

Environments, value-stores, flow-sets, and closures simulate under the straightforward structural liftings we would expect given their identical layout.

$$
\begin{aligned}
\hat{\rho} \widehat{\sqsupseteq}_{Env} \tilde{\rho} &\iff \forall x \in dom(\tilde{\rho}).\ \hat{\rho}(x) \widehat{\sqsupseteq}_{Env} \tilde{\rho}(x)\\
\hat{\sigma} \widehat{\sqsupseteq}_{Store} \tilde{\sigma} &\iff \forall \hat{a} \equiv \tilde{a}.\ \hat{\sigma}(\hat{a}) \widehat{\sqsupseteq}_D \tilde{\sigma}(\tilde{a})\\
\hat{d} \widehat{\sqsupseteq}_D \tilde{d} &\iff \forall \widetilde{clo} \in \tilde{d}.\ \exists \widehat{clo} \in \hat{d}.\ \widehat{clo} \widehat{\sqsupseteq}_{Clo} \widetilde{clo}\\
(lam, \hat{\rho}) \widehat{\sqsupseteq}_{Clo} (lam, \tilde{\rho}) &\iff \hat{\rho} \widehat{\sqsupseteq}_{Env} \tilde{\rho}
\end{aligned}
$$

As thoughout, the use of lam twice in the above definition indicates these components are exactly equal.

In the general case, we assume a user-provided notion of congruence for addresses produced by the standard allocator:

Assumption 1 (Allocation)**.** *The allocators $\widehat{alloc}$ and $\widetilde{alloc}$ produce congruent addresses if the input states simulate.*

$$\hat{\varsigma} \widehat{\sqsupseteq}_\Sigma \tilde{\varsigma} \implies \widehat{alloc}(x, \hat{\varsigma}) \equiv \widetilde{alloc}(x, \tilde{\varsigma})$$

In the case of both the 0-CFA and 1-CFA-style allocators we have provided, this is trivial to show as addresses are produced from a pairing of syntactic components which may be compared for exact equality.

Note that for power-set domains (e.g. $\hat{D}$) our nesting of quantification is ordered so that every element in our finite-state model must have a representative (e.g. a closure which is at-most-as-precise) in the infinite-state model. Simulation for sets of reachable configurations is inherantly one-to-many: a set of exact stacks is simulated by a single continuation-store address. More precisely, a finite state $(e, \tilde{\rho}, \tilde{a}_\kappa) \in \tilde{r}$ is simulated by the (possibly infinite) set of all pushdown states unique to the expression e, and any environment $\hat{\rho}$ which simulates $\tilde{\rho}$. We define this set of states using a helper $\mathcal{X}_{\hat{\xi}}(\hat{c})$ which returns an analysis result (some $\hat{\xi}_c$) containing only the partition of reachable states containing $\hat{c}$:

$$\mathcal{X}_{(\hat{r},\hat{\sigma})}((e, \hat{\rho}, \hat{\kappa})) = (\{(e, \hat{\rho}, \hat{\kappa}') \mid (e, \hat{\rho}, \hat{\kappa}') \in \hat{r}\}, \hat{\sigma})$$

We also define a transition relation ($\widehat{\rightsquigarrow}_\mathcal{X}$) for these partitioned, widened analyses such that:

$$
\begin{aligned}
&\mathcal{X}_{(\hat{r},\hat{\sigma})}((e, \hat{\rho}, \hat{\kappa})) \widehat{\rightsquigarrow}_\mathcal{X} \mathcal{X}_{(\hat{r}',\hat{\sigma}')}((e', \hat{\rho}', \hat{\kappa}')) \iff\\
&\qquad (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \widehat{\rightsquigarrow}_\Sigma (e', \hat{\rho}', \hat{\sigma}'', \hat{\kappa}') \wedge (\hat{r}, \hat{\sigma}) \widehat{\rightsquigarrow}_\Xi (\hat{r}', \hat{\sigma}')
\end{aligned}
$$

We throw away partially incremented value-stores (each $\hat{\sigma}''$) as the transition ($\widehat{\rightsquigarrow}_\Xi$) already defines $\hat{\sigma}'$ and it may be greater than the least-upper-bound of all $\hat{\sigma}''$ in any given partition.

A partition of a pushdown result simulates a single finite state when their expressions and environments simulate:

$$
\begin{aligned}
&\overbrace{\mathcal{X}_{(\hat{r},\hat{\sigma})}((e, \hat{\rho}, \hat{\kappa}))}^{(\hat{r}',\hat{\sigma})} \widehat{\sqsupseteq}_\mathcal{X} (e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa) \iff\\
&\hat{\sigma} \widehat{\sqsupseteq}_{Store} \tilde{\sigma} \wedge \hat{\rho} \widehat{\sqsupseteq}_{Env} \tilde{\rho} \wedge \{\hat{\kappa} \mid (e, \hat{\rho}, \hat{\kappa}) \in \hat{r}\} \widehat{\sqsupseteq}_{Kont} \tilde{a}_\kappa \text{ (via } \tilde{\sigma}_\kappa)
\end{aligned}
$$

Simulation for widened finite state analyses is then defined in terms of simulation for individual finite states:

$$
\begin{aligned}
&(\hat{r}, \hat{\sigma}) \widehat{\sqsupseteq}_\Xi (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa) \iff \hat{\sigma} \widehat{\sqsupseteq}_{Store} \tilde{\sigma} \wedge\\
&\forall (e, \tilde{\rho}, \tilde{a}_\kappa) \in \tilde{r}.\ \exists \hat{c} \in \hat{r}.\ \mathcal{X}_{(\hat{r},\hat{\sigma})}(\hat{c}) \widehat{\sqsupseteq}_\mathcal{X} (e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa)
\end{aligned}
$$

Finally the heart of this connection, stack simulation, may be defined recursively. The empty stack simulates the halt address in any store:

$$\{\epsilon\} \widehat{\sqsupseteq}_{Kont} \tilde{a}_{\text{halt}} \text{ (via } \tilde{\sigma}_\kappa)$$

Non-empty stacks simulate by the following:

$$
\begin{aligned}
&\hat{k} \widehat{\sqsupseteq}_{Kont} \tilde{a}_\kappa \text{ (via } \tilde{\sigma}_\kappa) \iff\\
&\forall ((x, e, \tilde{\rho}), \tilde{a}'_\kappa) \in \tilde{\sigma}_\kappa(\tilde{a}_\kappa).\ \exists ((x, e, \hat{\rho}) : \hat{\kappa}) \in \hat{k}.\ \hat{\rho} \widehat{\sqsupseteq}_{Env} \tilde{\rho} \wedge\\
&\qquad \{\hat{\kappa} \mid ((x, e, \hat{\rho}) : \hat{\kappa}) \in \hat{k}\} \widehat{\sqsupseteq}_{Kont} \tilde{a}'_\kappa \text{ (via } \tilde{\sigma}_\kappa)
\end{aligned}
$$

5.1 Preliminary properties

We are now ready to describe some properties of this connection that make it easier to show our modeling of continuations is as precise as the unbounded-stack machine. We present these without discussion as they are straightforward.

Lemma 2 (Atomic-evaluation)**.** *The results of atomic-evaluation simulate whenever their arguments do.*

$$\hat{\rho} \widehat{\sqsupseteq}_{Env} \tilde{\rho} \wedge \hat{\sigma} \widehat{\sqsupseteq}_{Store} \tilde{\sigma} \implies \hat{\mathcal{A}}(\text{æ}, \hat{\rho}, \hat{\sigma}) \widehat{\sqsupseteq}_D \tilde{\mathcal{A}}(\text{æ}, \tilde{\rho}, \tilde{\sigma})$$

Proof. By cases on $\mathcal{A}$. □

Lemma 3 (Flow-set-join)**.** *The least-upper-bound of flow-sets will itself simulate the join of flow-sets these simulate.*

$$\hat{d}_1 \widehat{\sqsupseteq}_D \tilde{d}_1 \wedge \hat{d}_2 \widehat{\sqsupseteq}_D \tilde{d}_2 \implies (\hat{d}_1 \cup \hat{d}_2) \widehat{\sqsupseteq}_D (\tilde{d}_1 \cup \tilde{d}_2)$$

Proof. From the definition of ($\widehat{\sqsupseteq}_D$). □

Lemma 4 (Store-join)**.** *The least-upper-bound of stores will itself simulate the join of the stores which these simulate.*

$$\hat{\sigma}_1 \widehat{\sqsupseteq}_{Store} \tilde{\sigma}_1 \wedge \hat{\sigma}_2 \widehat{\sqsupseteq}_{Store} \tilde{\sigma}_2 \implies (\hat{\sigma}_1 \sqcup \hat{\sigma}_2) \widehat{\sqsupseteq}_{Store} (\tilde{\sigma}_1 \sqcup \tilde{\sigma}_2)$$

Proof. A point-wise lifting of Lemma 3. □

Lemma 5 (Base-case)**.** *Each abstract interpretation begins as a simulation of the other.*

Proof. Specifically that:

$$(\{(e_0, \varnothing, \epsilon)\}, \bot) \sqsupseteq_\Xi (\{(e_0, \varnothing, \tilde{a}_{\text{halt}})\}, \bot, \bot)$$

Falls out from the definitions given. □

5.2 Central lemmas and theorems

This section structures a proof of bisimulation between the unbounded-stack analysis and our optimized finite-state analysis. We will focus exclusively on details of showing the precision of the finite-state result.

Theorem 6 (Bisimulation)**.** *Our abstract interpretation with a precise stack and our abstract interpretation with precise (target-state) continuation allocation precisely simulate one another.*

Proof. By the induction argument of Lemmas 5, 7, and 8. □

While we omit the details of the soundness half of this bisimulation proof, it directly follows from the AAM methodology [12].

Lemma 7 (Soundness)**.** *An inductive step shows that soundness is preserved across transition:*

$$\hat{\xi} \sqsubseteq_\Xi \tilde{\xi} \;\wedge\; \tilde{\xi} \rightsquigarrow_\Xi^\sim \tilde{\xi}' \;\wedge\; \hat{\xi} \rightsquigarrow_\Xi^\wedge \hat{\xi}' \implies \hat{\xi}' \sqsubseteq_\Xi \tilde{\xi}'$$

Proof. By cases; the reverse of Lemma 8. □

Now we come to the lemma in which we prove the precision of our analysis.

Lemma 8 (Precision)**.** *An inductive step shows that precision is preserved across store-widened transition.*

$$\hat{\xi} \sqsupseteq_\Xi \tilde{\xi} \;\wedge\; \tilde{\xi} \rightsquigarrow_\Xi^\sim \tilde{\xi}' \;\wedge\; \hat{\xi} \rightsquigarrow_\Xi^\wedge \hat{\xi}' \implies \hat{\xi}' \sqsupseteq_\Xi \tilde{\xi}'$$

Proof. Divided into Lemmas 9 and 10. □

We divide this lemma into two parts. The first part shows that the precision of the globally-widened store is maintained across the transition $(\rightsquigarrow_\Xi^\sim)$. We omit this proof as it is very similar to the second part which more clearly focuses on the key ideas.

Lemma 9 (Store-Precision)**.** *An inductive step shows that global-store precision is preserved across transition.*

$$\hat{\xi} \sqsupseteq_\Xi \tilde{\xi} \;\wedge\; \tilde{\xi} \rightsquigarrow_\Xi^\sim \overbrace{(\tilde{r}', \tilde{\sigma}', \tilde{\sigma}'_\kappa)}^{\tilde{\xi}'} \wedge \hat{\xi} \rightsquigarrow_\Xi^\wedge \overbrace{(\hat{r}', \hat{\sigma}')}^{\hat{\xi}'} \implies \hat{\sigma}' \sqsupseteq_{\mathit{Store}} \tilde{\sigma}'$$

Proof. Worked in a similar manner as Lemma 10. □

Finally, we arrive at the crucial aspect of proving the precision of our optimized finite-state analysis. This lemma shows that the precision of reachable configurations and stacks is maintained across the transition $(\rightsquigarrow_\Xi^\sim)$.

Lemma 10 (Reachability/Stack-Precision)**.** *An inductive step shows that every $(e', \tilde{\rho}', \tilde{a}'_\kappa)$ reachable across a widened finite-state transition is accounted for by an $(e', \hat{\rho}', \hat{\kappa}')$ reachable across a widened pushdown transition such that the partition containing $(e', \hat{\rho}', \hat{\kappa}')$ accounts for every stack implied by $\tilde{a}'_\kappa$.*

$$\begin{aligned}\hat{\xi} \sqsupseteq_\Xi \tilde{\xi} \;\wedge\; \overbrace{(\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa)}^{\tilde{\xi}} \rightsquigarrow_\Xi^\sim \overbrace{(\tilde{r}', \tilde{\sigma}', \tilde{\sigma}'_\kappa)}^{\tilde{\xi}'} \wedge \overbrace{(\hat{r}, \hat{\sigma})}^{\hat{\xi}} \rightsquigarrow_\Xi^\wedge \overbrace{(\hat{r}', \hat{\sigma}')}^{\hat{\xi}'} \\ \implies \forall (e', \tilde{\rho}', \tilde{a}'_\kappa) \in \tilde{r}'.\, \exists (e', \hat{\rho}', \hat{\kappa}') \in \hat{r}'. \\ \mathcal{X}_{(\hat{r}', \hat{\sigma}')}((e', \hat{\rho}', \hat{\kappa}')) \sqsupseteq_\mathcal{X} (e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}'_\kappa)\end{aligned}$$

Proof. Without loss of generality, we select a $(e', \tilde{\rho}', \tilde{a}'_\kappa) \in \tilde{r}'$. From the antecedent $\tilde{\xi} \rightsquigarrow_\Xi^\sim \tilde{\xi}'$ and the definition of $(\rightsquigarrow_\Xi^\sim)$, there exist $\tilde{\varsigma} = (e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa)$ and $\tilde{\varsigma}' = (e', \tilde{\rho}', \tilde{\sigma}'', \tilde{\sigma}''_\kappa, \tilde{a}'_\kappa)$ for some $\tilde{\sigma}''$ and $\tilde{\sigma}''_\kappa$ such that $\tilde{\varsigma} \rightsquigarrow_\Sigma^\sim \tilde{\varsigma}'$. From the antecedent $\hat{\xi} \sqsupseteq_\Xi \tilde{\xi}$, we know there exists a $(e, \hat{\rho}, \hat{\kappa}) \in \hat{r}$ such that $\mathcal{X}_{(\hat{r}, \hat{\sigma})}((e, \hat{\rho}, \hat{\kappa})) \sqsupseteq_\mathcal{X} (e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa)$. We must show that:

$$\exists (e', \hat{\rho}', \hat{\kappa}') \in \hat{r}'.\, \mathcal{X}_{(\hat{r}', \hat{\sigma}')}((e', \hat{\rho}', \hat{\kappa}')) \sqsupseteq_\mathcal{X} (e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}'_\kappa)$$

For this, it is sufficient to show that:

$$\mathcal{X}_{(\hat{r}, \hat{\sigma})}((e, \hat{\rho}, \hat{\kappa})) \rightsquigarrow_\mathcal{X}^\wedge \mathcal{X}_{(\hat{r}', \hat{\sigma}')}((e', \hat{\rho}', \hat{\kappa}'))$$

and

$$\mathcal{X}_{(\hat{r}', \hat{\sigma}')}((e', \hat{\rho}', \hat{\kappa}')) \sqsupseteq_\mathcal{X} (e', \tilde{\rho}', \tilde{\sigma}'', \tilde{\sigma}''_\kappa, \tilde{a}'_\kappa).$$

Diagrammatically:

$$\begin{array}{ccc} \overbrace{(e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa)}^{\tilde{\xi}} & \xrightarrow{\rightsquigarrow_\Sigma^\sim} & \overbrace{(e', \tilde{\rho}', \tilde{\sigma}'', \tilde{\sigma}''_\kappa, \tilde{a}'_\kappa)}^{\tilde{\xi}'} \\ \sqsupseteq_\mathcal{X} \uparrow & & \uparrow \sqsupseteq_\mathcal{X} \\ \underbrace{\mathcal{X}_{(\hat{r}, \hat{\sigma})}((e, \hat{\rho}, \hat{\kappa}))}_{\hat{\xi}_e} & \xrightarrow{\rightsquigarrow_\mathcal{X}^\wedge} & \underbrace{\mathcal{X}_{(\hat{r}', \hat{\sigma}')}((e', \hat{\rho}', \hat{\kappa}'))}_{\hat{\xi}'_e} \end{array}$$

This transition $(\rightsquigarrow_\mathcal{X}^\wedge)$ is defined:

$$(e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \rightsquigarrow_\Sigma^\wedge (e', \hat{\rho}', \hat{\sigma}'', \hat{\kappa}') \;\wedge\; (\hat{r}, \hat{\sigma}) \rightsquigarrow_\Xi^\wedge (\hat{r}', \hat{\sigma}')$$

The second conjunct is assumed as part of our antecedent. Thus it is sufficient to show there exists a $(e', \hat{\rho}', \hat{\kappa}') \in \hat{r}'$ such that $(e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \rightsquigarrow_\Sigma^\wedge (e', \hat{\rho}', \hat{\sigma}'', \hat{\kappa}')$, for some $\hat{\sigma}''$, and

$$\mathcal{X}_{(\hat{r}', \hat{\sigma}')}((e', \hat{\rho}', \hat{\kappa}')) \sqsupseteq_\mathcal{X} (e', \tilde{\rho}', \tilde{\sigma}'', \tilde{\sigma}''_\kappa, \tilde{a}'_\kappa).$$

For this, we break into cases on the expression e.

[Call] In this case our antecedent $\tilde{\varsigma} \rightsquigarrow_\Sigma^\sim \tilde{\varsigma}'$ expands to:

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

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

Then the transition rule $\hat{\varsigma} \rightsquigarrow_\Sigma^\wedge \hat{\varsigma}'$ expands to:

$\overbrace{(\texttt{(let ([}x\ \texttt{(}f\ \text{æ}\texttt{)])}\ e), \hat{\rho}, \hat{\sigma}, \hat{\kappa})}^{\hat{\xi}} \rightsquigarrow_\Sigma^\wedge \overbrace{(e', \hat{\rho}', \hat{\sigma}', \hat{\phi} : \hat{\kappa})}^{\hat{\xi}'}$, where

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

We fix a $(\texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e'\texttt{)}, \tilde{\rho}_\lambda) \in \tilde{\mathcal{A}}(f, \tilde{\rho}, \tilde{\sigma})$ without loss of generality. Then from Lemma 2 we know a simulating closure exists:

$$\exists (\texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e'\texttt{)}, \hat{\rho}_\lambda) \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) : \hat{\rho}_\lambda \sqsupseteq_{\mathit{Env}} \tilde{\rho}_\lambda$$

Proving our simulation decomposes into showing that:

$$\hat{\sigma}' \sqsupseteq_{\mathit{Store}} \tilde{\sigma}' \wedge \hat{\rho}' \sqsupseteq_{\mathit{Env}} \tilde{\rho}' \wedge \{\hat{\kappa}'' \mid (e', \hat{\rho}', \hat{\kappa}'') \in \hat{r}'\} \sqsupseteq_{\mathit{Kont}} \tilde{a}'_\kappa \text{ (via } \tilde{\sigma}'_\kappa)$$

Thus by Lemma 9 the first conjunct of our goal is satisfied ($\hat{\sigma}' \sqsupseteq_{Store} \tilde{\sigma}'$), and by Assumption 1 the second conjunct of our goal is satisfied ($\hat{\rho}' \sqsupseteq_{Env} \tilde{\rho}'$). Our antecedent simulation also tells us that every stack implied by $\tilde{a}_\kappa$ exists as $\hat{\kappa}$ in some $\hat{\varsigma}$.

$$\{\hat{\kappa}'' \mid (e, \hat{\rho}, \hat{\kappa}'') \in \hat{r}\} \sqsupseteq_{Kont} \tilde{a}_\kappa \text{ (via } \tilde{\sigma}_\kappa)$$

From here we fix an address in the continuation-store and look at which stacks may be contributed to it across our selected $\tilde{\varsigma} \rightsquigarrow^{\tilde{}}_{\Sigma} \tilde{\varsigma}'$. We must ensure that every such stack is accounted for in some transition $\hat{\varsigma} \rightsquigarrow^{\hat{}}_{\Sigma} \hat{\varsigma}'$ – and here we encounter the *crucial* moment which differentiates our technique from overly-precise AAC and previous imprecise allocation policies.

Addresses in $\tilde{a}'_\kappa$, allocated across $\tilde{\varsigma} \rightsquigarrow^{\tilde{}}_{\Sigma} \tilde{\varsigma}'$, are of the form $(e', \tilde{\rho}')$ and are specific to those components of our target state. This means we are able to assume a $\tilde{\varsigma}$ which preceeds this expression and environment. As exact stacks built across our $\hat{\varsigma} \rightsquigarrow^{\hat{}}_{\Sigma} \hat{\varsigma}'$ must always be unique to e' and $\hat{\rho}'$, we may consider the cases where $\hat{\rho}' \sqsupseteq_{Env} \tilde{\rho}'$ and conclude that every stack implied by $\tilde{\varsigma}'$ is accounted for. If instead we were to use the more common choice of e or e' when allocating a continuation (as is also the case for monovariant continuation-passing-style analyses), we would need to consider transitions $\tilde{\varsigma}_{\text{other}} \rightsquigarrow^{\tilde{}}_{\Sigma} \ldots$ which are not at least specific to our chosen $\tilde{\rho}'$ and show they too may only contribute implicit stacks accounted for by our transitions to pushdown states unique to both e' and $\hat{\rho}'$. This is clearly not true in the general case.

AAC on the other hand uses the entire source configuration (at least e and $\tilde{\rho}$) which is at least specific to our selected e' and $\tilde{\rho}'$ because predecessors determine these components of their successors uniquely. This is why AAC is as precise as our technique (and as precise as PDCFA); however, because it may be *more* specific then simply e' and $\tilde{\rho}'$ (various $e, \tilde{\rho}$ may preceed $e', \tilde{\rho}'$), AAC may differentiate stacks without any justifying further benefit to precision. We may observe here how selecting the target state is the minimal choice for obtaining perfect precision.

The continuation address we have fixed therefore is $(e', \tilde{\rho}')$ and we may also fix our contributed continuation $((x, e, \tilde{\rho}), \tilde{a}_\kappa) \in \tilde{\sigma}'_\kappa((e', \tilde{\rho}'))$ and demonstrate its simulation:

$$\exists((x, e, \hat{\rho}) : \hat{\kappa}) \in \overbrace{K_{(e', \hat{\rho}', \_) \in \hat{r}'}}^{\hat{k}} : \hat{\rho} \sqsupseteq_{Env} \tilde{\rho} \wedge K_{((x,e,\hat{\rho}):\_) \in \hat{k}} \sqsupseteq_{Kont} \tilde{a}_\kappa \text{ (via } \tilde{\sigma}'_\kappa)$$

Which falls out immediately from $\hat{\varsigma} \rightsquigarrow^{\hat{}}_{\Sigma} \hat{\varsigma}'$ and our prior simulation for $\tilde{a}_\kappa$. This may be restated as our desired consequent:

$$K_{(e', \hat{\rho}', \_) \in \hat{r}'} \sqsupseteq_{Kont} \tilde{a}'_\kappa \text{ (via } \tilde{\sigma}'_\kappa)$$

[Return] In this case our antecedent $\tilde{\varsigma} \rightsquigarrow^{\tilde{}}_{\Sigma} \tilde{\varsigma}'$ expands to:

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

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

Then the transition rule $\hat{\varsigma} \rightsquigarrow^{\hat{}}_{\Sigma} \hat{\varsigma}'$ expands to:

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

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

We unroll our simulation for $\tilde{a}_\kappa$ a single step and obtain:

$$\begin{aligned} &\overbrace{K_{(e, \hat{\rho}, \_) \in \hat{r}}}^{\hat{k}} \sqsupseteq_{Kont} \tilde{a}_\kappa \text{(via } \tilde{\sigma}_\kappa) \implies \\ &\forall((x, e', \tilde{\rho}_\kappa), \tilde{a}'_\kappa) \in \tilde{\sigma}_\kappa(\tilde{a}_\kappa).\ \exists((x, e', \hat{\rho}_\kappa) : \hat{\kappa}') \in \hat{k} : \\ &\quad \hat{\rho}_\kappa \sqsupseteq_{Env} \tilde{\rho}_\kappa \wedge K_{((x,e',\hat{\rho}_\kappa):\_) \in \hat{k}} \sqsupseteq_{Kont} \tilde{a}'_\kappa \text{ (via } \tilde{\sigma}_\kappa) \end{aligned}$$

We fix $((x, e', \tilde{\rho}_\kappa), \tilde{a}'_\kappa) \in \tilde{\sigma}_\kappa(\tilde{a}_\kappa)$ without loss of generality. This is a specific stack frame being popped and its tail $\tilde{a}'_\kappa$. As our successor $\tilde{\varsigma}'$ is reached at every stack *implicit* in this tail address, we must show that likewise every such stack is reached explicitly in some successor to $\hat{\varsigma}'$.

Those successors are derived from all $((x, e', \hat{\rho}_\kappa) : \hat{\kappa}') \in \hat{k}$. We know from the existence of $\tilde{\varsigma}'$ that at least one such top frame exists. We also know from $\hat{\rho}_\kappa \sqsupseteq_{Env} \tilde{\rho}_\kappa$ and Lemma 1 that $\hat{\rho}' \sqsupseteq_{Env} \tilde{\rho}'$. From our unrolled simulation for $\tilde{a}_\kappa$ above, we find that the total set of $\hat{\kappa}'$ values transitioned to is a simulation of $\tilde{a}'_\kappa$.

$$\begin{aligned} K_{((x,e',\hat{\rho}_\kappa):\_) \in \hat{k}} \sqsupseteq_{Kont} \tilde{a}'_\kappa \text{ (via } \tilde{\sigma}_\kappa) &\implies \\ K_{(e', \hat{\rho}', \_) \in \hat{r}'} \sqsupseteq_{Kont} \tilde{a}'_\kappa \text{ (via } \tilde{\sigma}'_\kappa)& \end{aligned}$$

In both these cases our scheme preserves precision. □

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 the corresponding function application. Recent approaches have made significant progress in finally addressing this problem, however each has suffered from serious drawbacks. PDCFA suffers from a substantial added engineering cost on top of the labor requirements innate in its underlying analysis. In addition, it causes a quadratic-factor increase in the run-time complexity of analysis. AAC is trival to implement for frameworks using store-allocated continuations, but incurs an even worse increase in run-time complexity.

We have presented an approach which synthesizes the lessons learned from both PDCFA and AAC, yielding a technique which is both simple to implement and adds no asymptotic cost to analysis complexity. PDCFA's representation for sets of continuations uses the minimal differentiation needed to obtain perfect stack-precision, but it requires these sets to be recomputed unnecessarily for each intraprocedural configuration in an abstract function invocation. AAC requires no significant engineering cost, and its continuation addresses propagate through an abstract function invocation without needing to be recomputed; however, the addresses it selects are more precise than is necessary and it incurs a large overhead without further benefit to analysis precision.

Putting these ideas together, we have shown that the ideal continuation address to use is simply the polyvariant entry-point of a function: its expression and abstract binding-environment. This choice of continuation address yields a finite-state analysis whose call transitions are precisely matched with return transitions, but without any asymptotic run-time overhead.

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 Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977. ACM Press, New York.

[3] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, TX, 1979. ACM Press, New York.

[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 *International Conference on Functional Programming*, pages 177–188, September 2012.

[6] J. I. Johnson. AAC complexity analysis. Unpublished correspondence., 2015.

[7] J. I. Johnson and D. Van Horn. Abstracting abstract control. In *Proceedings of the ACM Symposium on Dynamic Languages*, October 2014 2014.

[8] J. I. Johnson, N. Labich, M. Might, and D. Van Horn. Optimizing abstract abstract machines. In *Proceedings of the International Conference on Functional Programming*, September 2013.

[9] J. Midtgaard. Control-flow analysis of functional programs. *ACM Computing Surveys*, 44(3):10:1–10:33, Jun 2012.

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

[11] M. Might. Abstract interpreters for free. In *Static Analysis Symposium*, pages 407–421, September 2010.

[12] D. Van Horn and M. Might. Abstracting abstract machines. In *International Conference on Functional Programming*, page 51, Sep 2010.

[13] D. Vardoulakis and O. Shivers. CFA2: a context-free approach to control-flow analysis. In *Proceedings of the European Symposium on Programming*, volume 6012, LNCS, pages 570–589, 2010.