Introspective Pushdown Analysis of Higher-Order Programs

Christopher Earl
University of Utah
cwearl@cs.utah.edu

Ilya Sergey
KU Leuven
ilya.sergey@cs.kuleuven.be

Matthew Might
University of Utah
might@cs.utah.edu

David Van Horn
Northeastern University
dvanhorn@ccs.neu.edu

Abstract

In the static analysis of functional programs, pushdown flow analysis and abstract garbage collection skirt just inside the boundaries of soundness and decidability. Alone, each method reduces analysis times and boosts precision by orders of magnitude. This work illuminates and conquers the theoretical challenges that stand in the way of combining the power of these techniques. The challenge in marrying these techniques is not subtle: computing the reachable control states of a pushdown system relies on limiting access during transition to the top of the stack; abstract garbage collection, on the other hand, needs full access to the entire stack to compute a root set, just as concrete collection does. *Introspective* pushdown systems resolve this conflict. Introspective pushdown systems provide enough access to the stack to allow abstract garbage collection, but they remain restricted enough to compute control-state reachability, thereby enabling the sound and precise product of pushdown analysis and abstract garbage collection. Experiments reveal synergistic interplay between the techniques, and the fusion demonstrates "better-than-both-worlds" precision.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Processors—Optimization; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis, Operational semantics

General Terms Languages, Theory

Keywords CFA2, pushdown systems, abstract interpretation, pushdown analysis, program analysis, abstract machines, abstract garbage collection, higher-order languages

1. Introduction

The recent development of a context-free[1] approach to control-flow analysis (CFA2) by Vardoulakis and Shivers has provoked a seismic shift in the static analysis of higher-order programs [22]. Prior to CFA2, a precise analysis of recursive behavior had been a stumbling block—even though flow analyses have an important role to play in optimization for functional languages, such as flow-driven inlining [13], interprocedural constant propagation [19] and type-check elimination [23].

While it had been possible to statically analyze recursion *soundly*, CFA2 made it possible to analyze recursion *precisely* by matching calls and returns without approximation. In its pursuit of recursion, clever engineering steered CFA2 just shy of undecidability. The payoff is an order-of-magnitude reduction in analysis time and an order-of-magnitude increase in precision.

For a visual measure of the impact, Figure 1 renders the abstract transition graph (a model of all possible traces through the program) for the toy program in Figure 2. For this example, pushdown analysis eliminates spurious return-flow from the use of recursion. But, recursion is just one problem of many for flow analysis. For instance, pushdown analysis still gets tripped up by the spurious cross-flow problem; at calls to `(id f)` and `(id g)` in the previous example, it thinks `(id g)` could be `f` *or* `g`.

Powerful techniques such as abstract garbage collection [14] were developed to solve the cross-flow problem.[2] In fact, abstract garbage collection, by itself, also delivers orders-of-magnitude improvements to analytic speed and precision. (See Figure 1 again for a visualization of that impact.)

It is natural to ask: can abstract garbage collection and pushdown anlysis work together? Can their strengths be multiplied? At first glance, the answer appears to be a disheartening *No*.

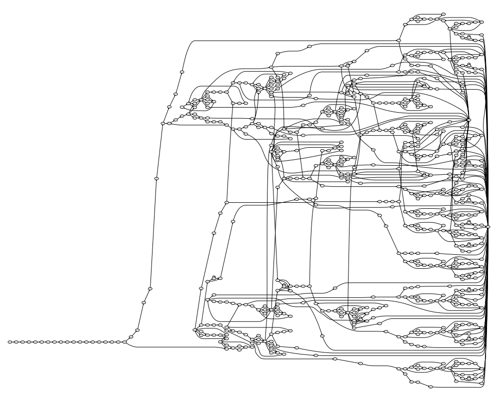
1.1 The problem: The whole stack *versus* just the top

Abstract garbage collections seems to require more than pushdown analysis can decidably provide: access to the full stack. Abstract garbage collection, like its name implies, discards unreachable values from an abstract store during the analysis. Like concrete garbage collection, abstract garbage collection also begins its sweep with a root set, and like concrete garbage collection, it must traverse the abstract stack to compute that root set. But, pushdown

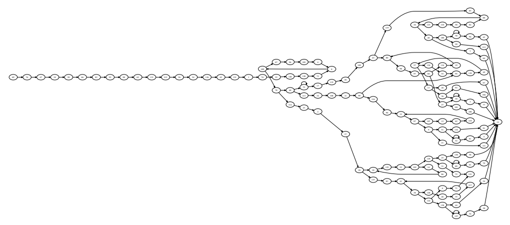
[1] As in context-free language, not context-sensitivity.

[2] The cross-flow problem arises because monotonicity prevents revoking a judgment like "procedure `f` flows to `x`," or "procedure `g` flows to `x`," once it's been made.

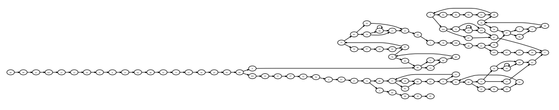
ICFP '12 September 10–12, Copenhagen, Denmark.
Copyright © 2012 ACM 978-1-4503-1054-3/12/09. . . $15.00



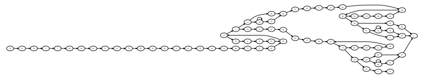
(1) without pushdown analysis or abstract GC: 653 states



(2) with pushdown only: 139 states



(3) with GC only: 105 states



(4) with pushdown analysis and abstract GC: 77 states

Figure 1. We generated an abstract transition graph for the same program from Figure 2 four times: (1) without pushdown analysis or abstract garbage collection; (2) with only abstract garbage collection; (3) with only pushdown analysis; (4) with both pushdown analysis and abstract garbage collection. With only pushdown or abstract GC, the abstract transition graph shrinks by an order of magnitude, but in different ways. The pushdown-only analysis is confused by variables that are bound to several different higher-order functions, but for short durations. The abstract-GC-only is confused by non-tail-recursive loop structure. With both techniques enabled, the graph shrinks by nearly half yet again and fully recovers the control structure of the original program.

```
(define (id x) x)

(define (f n)
  (cond [(<= n 1)  1]
        [else      (* n (f (- n 1)))]))

(define (g n)
  (cond [(<= n 1)  1]
        [else      (+ (* n n) (g (- n 1)))]))

(print (+ ((id f) 3) ((id g) 4)))
```

Figure 2. A small example to illuminate the strengths and weaknesses of both pushdown analysis and abstract garbage collection.

systems are restricted to viewing the top of the stack (or a bounded depth)—a condition violated by this traversal.

Fortunately, abstract garbage collection does not need to arbitrarily modify the stack. In fact, it does not even need to know the order of the frames; it only needs the *set* of frames on the stack. We find a richer class of machine—*introspective* pushdown systems—which retains just enough restrictions to compute reachable control states, yet few enough to enable abstract garbage collection.

It is therefore possible to fuse the full benefits of abstract garbage collection with pushdown analysis. The dramatic reduction in abstract transition graph size from the top to the bottom in Figure 1 (and echoed by later benchmarks) conveys the impact of this fusion.

Secondary motivations There are three strong secondary motivations for this work: (1) bringing context-sensitivity to pushdown analysis; (2) exposing the context-freedom of the analysis; and (3) enabling pushdown analysis without continuation passing style.

In CFA2, monovariant (0CFA-like) context-sensitivity is etched directly into the abstract semantics, which are in turn, phrased in terms of an explicit (imperative) summarization algorithm for a partitioned continuation-passing style.

In addition, the context-freedom of the analysis is buried implicitly inside this algorithm. No pushdown system or context-free grammar is explicitly identified. A necessary precursor to our work was to make the pushdown system in CFA2 explicit.

A third motivation was to show that a transformation to continuation-passing style is unnecessary for pushdown analysis. In fact, pushdown analysis is arguably more natural over direct-style programs.

1.2 Overview

We first review preliminaries to set a consistent feel for terminology and notation, particularly with respect to pushdown systems. The derivation of the analysis begins with a concrete CESK-machine-style semantics for A-Normal Form λ-calculus. The next step is an infinite-state abstract interpretation, constructed by bounding the C(ontrol), E(nvironment) and S(tore) portions of the machine while leaving the stack—the K(ontinuation)—unbounded. A simple shift in perspective reveals that this abstract interpretation is a rooted pushdown system.

We then introduce abstract garbage collection and quickly find that it violates the pushdown model with its traversals of the stack. To prove the decidability of control-state reachability, we formulate introspective pushdown systems, and recast abstract garbage collection within this framework. We then show that control-state reachability is decidable for introspective pushdown systems as well.

We conclude with an implementation and empirical evaluation that shows strong synergies between pushdown analysis and abstract garbage collection, including significant reductions in the size of the abstract state transition graph.

1.3 Contributions

We make the following contributions:

1. Our primary contribution is demonstrating the decidability of fusing abstract garbage collection with pushdown flow analysis of higher-order programs. Proof comes in the form of a fixed-point solution for computing the reachable control-states of an introspective pushdown system and an embedding of abstract garbage collection as an introspective pushdown system.
2. We show that classical notions of context-sensitivity, such as k-CFA and poly/CFA, have direct generalizations in a pushdown setting: monovariance[3] is *not* an essential restriction, as in CFA2.
3. We make the context-free aspect of CFA2 explicit: we clearly define and identify the pushdown system. We do so by starting with a classical CESK machine and systematically abstracting until a pushdown system emerges. We also remove the orthogonal frame-local-bindings aspect of CFA2, so as to directly solely on the pushdown nature of the analysis.
4. We remove the requirement for CPS-conversion by synthesizing the analysis directly for direct-style (in the form of A-normal form lambda-calculus).
5. We empirically validate claims of improved precision on a suite of benchmarks. We find synergies between pushdown analysis and abstract garbage collection that makes the whole greater that the sum of its parts.

2. Pushdown preliminaries

The literature contains many equivalent definitions of pushdown machines, so we adapt our own definitions from Sipser [20]. *Readers familiar with pushdown theory may wish to skip ahead.*

2.1 Syntactic sugar

When a triple (x, ℓ, x') is an edge in a labeled graph:

$$x \overset{\ell}{\rightarrowtail} x' \equiv (x, \ell, x').$$

Similarly, when a pair (x, x') is a graph edge:

$$x \rightarrowtail x' \equiv (x, x').$$

We use both string and vector notation for sequences:

$$a_1 a_2 \ldots a_n \equiv \langle a_1, a_2, \ldots, a_n \rangle \equiv \vec{a}.$$

2.2 Stack actions, stack change and stack manipulation

Stacks are sequences over a stack alphabet Γ. To reason about stack manipulation concisely, we first turn stack alphabets into "stack-action" sets; each character represents a change to the stack: push, pop or no change.

For each character γ in a stack alphabet Γ, the **stack-action** set $\Gamma_\pm$ contains a push character γ_+; a pop character γ_-; and a no-stack-change indicator, ϵ:

$$\begin{array}{rlll} g \in \Gamma_\pm ::= & \epsilon & & \text{[stack unchanged]} \\ \mid & \gamma_+ & \text{for each } \gamma \in \Gamma & [\text{pushed } \gamma] \\ \mid & \gamma_- & \text{for each } \gamma \in \Gamma & [\text{popped } \gamma]. \end{array}$$

In this paper, the symbol g represents some stack action.

When we develop introspective pushdown systems, we are going to need formalisms for easily manipulating stack-action strings and stacks. Given a string of stack actions, we can compact it into a minimal string describing net stack change. We do so through the operator $\lfloor \cdot \rfloor : \Gamma_\pm^* \to \Gamma_\pm^*$, which cancels out opposing adjacent push-pop stack actions:

$$\lfloor \vec{g}\, \gamma_+ \gamma_-\, \vec{g}\,' \rfloor = \lfloor \vec{g}\, \vec{g}\,' \rfloor \qquad \lfloor \vec{g}\, \epsilon\, \vec{g}\,' \rfloor = \lfloor \vec{g}\, \vec{g}\,' \rfloor,$$

so that $\lfloor \vec{g} \rfloor = \vec{g}$, if there are no cancellations to be made in the string $\vec{g}$.

We can convert a net string back into a stack by stripping off the push symbols with the stackify operator, $\lceil \cdot \rceil : \Gamma_\pm^* \rightharpoonup \Gamma^*$:

$$\lceil \gamma_+ \gamma'_+ \ldots \gamma_+^{(n)} \rceil = \langle \gamma^{(n)}, \ldots, \gamma', \gamma \rangle,$$

and for convenience, $[\vec{g}] = \lceil \lfloor \vec{g} \rfloor \rceil$. Notice the stackify operator is defined for strings containing only push actions.

2.3 Pushdown systems

A **pushdown system** is a triple $M = (Q, \Gamma, \delta)$ where:

1. Q is a finite set of control states;
2. Γ is a stack alphabet; and
3. $\delta \subseteq Q \times \Gamma_\pm \times Q$ is a transition relation.

The set $Q \times \Gamma^*$ is called the **configuration-space** of this pushdown system. We use $\mathbb{PDS}$ to denote the class of all pushdown systems.

For the following definitions, let $M = (Q, \Gamma, \delta)$.

- The labeled **transition relation** $(\longmapsto_M) \subseteq (Q \times \Gamma^*) \times \Gamma_\pm \times (Q \times \Gamma^*)$ determines whether one configuration may transition to another while performing the given stack action:

$$\begin{array}{lr} (q, \vec{\gamma}) \underset{M}{\overset{\epsilon}{\longmapsto}} (q', \vec{\gamma}) \text{ iff } q \overset{\epsilon}{\rightarrowtail} q' \in \delta & \text{[no change]} \\ (q, \gamma : \vec{\gamma}) \underset{M}{\overset{\gamma_-}{\longmapsto}} (q', \vec{\gamma}) \text{ iff } q \overset{\gamma_-}{\rightarrowtail} q' \in \delta & \text{[pop]} \\ (q, \vec{\gamma}) \underset{M}{\overset{\gamma_+}{\longmapsto}} (q', \gamma : \vec{\gamma}) \text{ iff } q \overset{\gamma_+}{\rightarrowtail} q' \in \delta & \text{[push].} \end{array}$$

- If unlabelled, the transition relation $(\longmapsto)$ checks whether *any* stack action can enable the transition:

$$c \underset{M}{\longmapsto} c' \text{ iff } c \underset{M}{\overset{g}{\longmapsto}} c' \text{ for some stack action } g.$$

- For a string of stack actions $g_1 \ldots g_n$:

$$c_0 \underset{M}{\overset{g_1 \cdots g_n}{\longmapsto}} c_n \text{ iff } c_0 \underset{M}{\overset{g_1}{\longmapsto}} c_1 \underset{M}{\overset{g_2}{\longmapsto}} \cdots \underset{M}{\overset{g_{n-1}}{\longmapsto}} c_{n-1} \underset{M}{\overset{g_n}{\longmapsto}} c_n,$$

for some configurations $c_0, \ldots, c_n$.

[3] Monovariance refers to an abstraction that groups all bindings to the same variable together: there is *one* abstract variant for all bindings to each variable.

- For the transitive closure:

$$c \underset{M}{\overset{*}{\longmapsto}} c' \text{ iff } c \underset{M}{\overset{\vec{g}}{\longmapsto}} c' \text{ for some action string } \vec{g}\ .$$

2.4 Rooted pushdown systems

A **rooted pushdown system** is a quadruple (Q, Γ, δ, q_0) in which (Q, Γ, δ) is a pushdown system and $q_0 \in Q$ is an initial (root) state. $\mathbb{RPDS}$ is the class of all rooted pushdown systems.

For a rooted pushdown system $M = (Q, \Gamma, \delta, q_0)$, we define the **reachable-from-root transition relation**:

$$c \underset{M}{\overset{g}{\longmapsto\!\!\!\rightarrow}} c' \text{ iff } (q_0, \langle\rangle) \underset{M}{\overset{*}{\longmapsto}} c \text{ and } c \underset{M}{\overset{g}{\longmapsto}} c'.$$

In other words, the root-reachable transition relation also makes sure that the root control state can actually reach the transition.

We overload the root-reachable transition relation to operate on control states:

$$q \underset{M}{\overset{g}{\longmapsto\!\!\!\rightarrow}} q' \text{ iff } (q, \vec{\gamma}) \underset{M}{\overset{g}{\longmapsto\!\!\!\rightarrow}} (q', \vec{\gamma}\,') \text{ for some stacks } \vec{\gamma}, \vec{\gamma}\,'.$$

For both root-reachable relations, if we elide the stack-action label, then, as in the un-rooted case, the transition holds if *there exists* some stack action that enables the transition:

$$q \underset{M}{\longmapsto\!\!\!\rightarrow} q' \text{ iff } q \underset{M}{\overset{g}{\longmapsto\!\!\!\rightarrow}} q' \text{ for some action } g.$$

2.5 Computing reachability in pushdown systems

A pushdown flow analysis can be construed as computing the *root-reachable* subset of control states in a rooted pushdown system, $M = (Q, \Gamma, \delta, q_0)$:

$$\left\{ q : q_0 \underset{M}{\longmapsto\!\!\!\rightarrow} q \right\}$$

Reps *et. al* and many others provide a straightforward "summarization" algorithm to compute this set [1, 8, 17, 18]. Our preliminary report also offers a reachability algorithm tailored to higher-order programs [4].

2.6 Nondeterministic finite automata

In this work, we will need a finite description of all possible stacks at a given control state within a rooted pushdown system. We will exploit the fact that the set of stacks at a given control point is a regular language. Specifically, we will extract a nondeterministic finite automaton accepting that language from the structure of a rooted pushdown system. A **nondeterministic finite automaton** (NFA) is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$:

- Q is a finite set of control states;
- Σ is an input alphabet;
- $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is a transition relation.
- q_0 is a distinguished start state.
- $F \subseteq Q$ is a set of accepting states.

We denote the class of all NFAs as $\mathbb{NFA}$.

3. Setting: A-Normal Form λ-calculus

Since our goal is analysis of *higher-order languages*, we operate on the λ-calculus. To simplify presentation of the concrete and abstract semantics, we choose A-Normal Form λ-calculus. (This is a strictly cosmetic choice: all of our results can be replayed *mutatis mutandis* in the standard direct-style setting as well.) ANF enforces an order of evaluation and it requires that all arguments to a function be atomic:

$$\begin{array}{rcll}
e \in \mathsf{Exp} & ::= & \texttt{(let ((}v\ \mathit{call}\texttt{))}\ e\texttt{)} & \text{[non-tail call]} \\
& \mid & \mathit{call} & \text{[tail call]} \\
& \mid & \text{æ} & \text{[return]} \\
f, \text{æ} \in \mathsf{Atom} & ::= & v \mid \mathit{lam} & \text{[atomic expressions]} \\
\mathit{lam} \in \mathsf{Lam} & ::= & \texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)} & \text{[lambda terms]} \\
\mathit{call} \in \mathsf{Call} & ::= & \texttt{(}f\ \text{æ}\texttt{)} & \text{[applications]} \\
v \in \mathsf{Var} & & \text{is a set of identifiers} & \text{[variables].}
\end{array}$$

We use the CESK machine of Felleisen and Friedman [5] to specify a small-step semantics for ANF. The CESK machine has an explicit stack, and under a structural abstraction, the stack component of this machine directly becomes the stack component of a pushdown system. The set of configurations (Conf) for this machine has the four expected components (Figure 3).

$$\begin{array}{rcll}
c \in \mathit{Conf} & = & \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont} & \text{[configurations]} \\
\rho \in \mathit{Env} & = & \mathsf{Var} \rightharpoonup \mathit{Addr} & \text{[environments]} \\
\sigma \in \mathit{Store} & = & \mathit{Addr} \to \mathit{Clo} & \text{[stores]} \\
\mathit{clo} \in \mathit{Clo} & = & \mathsf{Lam} \times \mathit{Env} & \text{[closures]} \\
\kappa \in \mathit{Kont} & = & \mathit{Frame}^{*} & \text{[continuations]} \\
\phi \in \mathit{Frame} & = & \mathsf{Var} \times \mathsf{Exp} \times \mathit{Env} & \text{[stack frames]} \\
a \in \mathit{Addr} & & \text{is an infinite set of addresses} & \text{[addresses].}
\end{array}$$

Figure 3. The concrete configuration-space.

3.1 Semantics

To define the semantics, we need five items:

1. $\mathcal{I} : \mathsf{Exp} \to \mathit{Conf}$ injects an expression into a configuration:
$$c_0 = \mathcal{I}(e) = (e, [], [], \langle\rangle).$$
2. $\mathcal{A} : \mathsf{Atom} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Clo}$ evaluates atomic expressions:
$$\begin{array}{rcll}
\mathcal{A}(\mathit{lam}, \rho, \sigma) & = & (\mathit{lam}, \rho) & \text{[closure creation]} \\
\mathcal{A}(v, \rho, \sigma) & = & \sigma(\rho(v)) & \text{[variable look-up].}
\end{array}$$
3. $(\Rightarrow) \subseteq \mathit{Conf} \times \mathit{Conf}$ transitions between configurations. (Defined below.)
4. $\mathcal{E} : \mathsf{Exp} \to \mathcal{P}(\mathit{Conf})$ computes the set of reachable machine configurations for a given program:
$$\mathcal{E}(e) = \{c : \mathcal{I}(e) \Rightarrow^{*} c\}\,.$$
5. $\mathit{alloc} : \mathsf{Var} \times \mathit{Conf} \to \mathit{Addr}$ chooses fresh store addresses for newly bound variables. The address-allocation function is an opaque parameter in this semantics, so that the forthcoming abstract semantics may also parameterize allocation. This parameterization provides the knob to tune the polyvariance and context-sensitivity of the resulting analysis. For the sake of defining the concrete semantics, letting addresses be natural numbers suffices, and then the allocator can choose the lowest

unused address:

$$Addr = \mathbb{N}$$
$$alloc(v, (e, \rho, \sigma, \kappa)) = 1 + \max(dom(\sigma)).$$

Transition relation To define the transition $c \Rightarrow c'$, we need three rules. The first rule handle tail calls by evaluating the function into a closure, evaluating the argument into a value and then moving to the body of the closure's λ-term:

$$\begin{aligned}
\overbrace{(\llbracket \texttt{(}f\ æ\texttt{)} \rrbracket, \rho, \sigma, \kappa)}^{c} &\Rightarrow \overbrace{(e, \rho'', \sigma', \kappa)}^{c'} \text{, where} \\
(\llbracket \texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)} \rrbracket, \rho') &= \mathcal{A}(f, \rho, \sigma) \\
a &= alloc(v, c) \\
\rho'' &= \rho'[v \mapsto a] \\
\sigma' &= \sigma[a \mapsto \mathcal{A}(æ, \rho, \sigma)].
\end{aligned}$$

Non-tail call pushes a frame onto the stack and evaluates the call:

$$\overbrace{(\llbracket \texttt{(let ((}v\ call\texttt{))}\ e\texttt{)} \rrbracket, \rho, \sigma, \kappa)}^{c} \Rightarrow \overbrace{(call, \rho, \sigma, (v, e, \rho) : \kappa)}^{c'}.$$

Function return pops a stack frame:

$$\begin{aligned}
\overbrace{(æ, \rho, \sigma, (v, e, \rho') : \kappa)}^{c} &\Rightarrow \overbrace{(e, \rho'', \sigma', \kappa)}^{c'} \text{, where} \\
a &= alloc(v, c) \\
\rho'' &= \rho'[v \mapsto a] \\
\sigma' &= \sigma[a \mapsto \mathcal{A}(æ, \rho, \sigma)].
\end{aligned}$$

4. Pushdown abstract interpretation

Our first step toward a static analysis is an abstract interpretation into an *infinite* state-space. To achieve a pushdown analysis, we simply abstract away less than we normally would. Specifically, we leave the stack height unbounded.

Figure 4 details the abstract configuration-space. To synthesize it, we force addresses to be a finite set, but crucially, we leave the stack untouched. When we compact the set of addresses into a finite set, the machine may run out of addresses to allocate, and when it does, the pigeon-hole principle will force multiple closures to reside at the same address. As a result, we have no choice but to force the range of the store to become a power set in the abstract configuration-space. The abstract transition relation has components analogous to those from the concrete semantics:

Program injection The abstract injection function $\hat{\mathcal{I}} : \mathsf{Exp} \to \widehat{Conf}$ pairs an expression with an empty environment, an empty store and an empty stack to create the initial abstract configuration:

$$\hat{c}_0 = \hat{\mathcal{I}}(e) = (e, [], [], \langle\rangle).$$

Atomic expression evaluation The abstract atomic expression evaluator, $\hat{\mathcal{A}} : \mathsf{Atom} \times \widehat{Env} \times \widehat{Store} \to \mathcal{P}(\widehat{Clo})$, returns the value of an atomic expression in the context of an environment and a store; it returns a *set* of abstract closures:

$$\begin{aligned}
\hat{\mathcal{A}}(lam, \hat{\rho}, \hat{\sigma}) &= \{(lam, \hat{\rho})\} && \text{[closure creation]} \\
\hat{\mathcal{A}}(v, \hat{\rho}, \hat{\sigma}) &= \hat{\sigma}(\hat{\rho}(v)) && \text{[variable look-up].}
\end{aligned}$$

$$\begin{aligned}
\hat{c} \in \widehat{Conf} &= \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \times \widehat{Kont} && \text{[configurations]} \\
\hat{\rho} \in \widehat{Env} &= \mathsf{Var} \rightharpoonup \widehat{Addr} && \text{[environments]} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \to \mathcal{P}\left(\widehat{Clo}\right) && \text{[stores]} \\
\widehat{clo} \in \widehat{Clo} &= \mathsf{Lam} \times \widehat{Env} && \text{[closures]} \\
\hat{\kappa} \in \widehat{Kont} &= \widehat{Frame}^{*} && \text{[continuations]} \\
\hat{\phi} \in \widehat{Frame} &= \mathsf{Var} \times \mathsf{Exp} \times \widehat{Env} && \text{[stack frames]} \\
\hat{a} \in \widehat{Addr} &\text{ is a } \textit{finite} \text{ set of addresses} && \text{[addresses].}
\end{aligned}$$

Figure 4. The abstract configuration-space.

Reachable configurations The abstract program evaluator $\hat{\mathcal{E}} : \mathsf{Exp} \to \mathcal{P}(\widehat{Conf})$ returns all of the configurations reachable from the initial configuration:

$$\hat{\mathcal{E}}(e) = \left\{\hat{c} : \hat{\mathcal{I}}(e) \leadsto^{*} \hat{c}\right\}.$$

Because there are an infinite number of abstract configurations, a naïve implementation of this function may not terminate.

Transition relation The abstract transition relation $(\leadsto) \subseteq \widehat{Conf} \times \widehat{Conf}$ has three rules, one of which has become non-deterministic. A tail call may fork because there could be multiple abstract closures that it is invoking:

$$\begin{aligned}
\overbrace{(\llbracket \texttt{(}f\ æ\texttt{)} \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\kappa})}^{\hat{c}} &\leadsto \overbrace{(e, \hat{\rho}'', \hat{\sigma}', \hat{\kappa})}^{\hat{c}'} \text{, where} \\
(\llbracket \texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)} \rrbracket, \hat{\rho}') &\in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \\
\hat{a} &= \widehat{alloc}(v, \hat{c}) \\
\hat{\rho}'' &= \hat{\rho}'[v \mapsto \hat{a}] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathcal{A}}(æ, \hat{\rho}, \hat{\sigma})].
\end{aligned}$$

We define all of the partial orders shortly, but for stores:

$$(\hat{\sigma} \sqcup \hat{\sigma}')(\hat{a}) = \hat{\sigma}(\hat{a}) \cup \hat{\sigma}'(\hat{a}).$$

A non-tail call pushes a frame onto the stack and evaluates the call:

$$\overbrace{(\llbracket \texttt{(let ((}v\ call\texttt{))}\ e\texttt{)} \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\kappa})}^{\hat{c}} \leadsto \overbrace{(call, \hat{\rho}, \hat{\sigma}, (v, e, \hat{\rho}) : \hat{\kappa})}^{\hat{c}'}.$$

A function return pops a stack frame:

$$\begin{aligned}
\overbrace{(æ, \hat{\rho}, \hat{\sigma}, (v, e, \hat{\rho}') : \hat{\kappa})}^{\hat{c}} &\leadsto \overbrace{(e, \hat{\rho}'', \hat{\sigma}', \hat{\kappa})}^{\hat{c}'} \text{, where} \\
\hat{a} &= \widehat{alloc}(v, \hat{c}) \\
\hat{\rho}'' &= \hat{\rho}'[v \mapsto \hat{a}] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathcal{A}}(æ, \hat{\rho}, \hat{\sigma})].
\end{aligned}$$

Allocation: Polyvariance and context-sensitivity In the abstract semantics, the abstract allocation function $\widehat{alloc} : \mathsf{Var} \times \widehat{Conf} \to \widehat{Addr}$ determines the polyvariance of the analysis. In a control-flow analysis, *polyvariance* literally refers to the number of abstract addresses (variants) there are for each variable. An advantage of this framework over CFA2 is that varying this abstract allocation function instantiates pushdown versions of classical flow analyses. All of the following allocation approaches can be used with the abstract semantics. The abstract allocation function is a parameter to the analysis.

Monovariance: Pushdown 0CFA Pushdown 0CFA uses variables themselves for abstract addresses:

$$\widehat{Addr} = \mathsf{Var}$$
$$alloc(v, \hat{c}) = v.$$

Context-sensitive: Pushdown 1CFA Pushdown 1CFA pairs the variable with the current expression to get an abstract address:

$$\widehat{Addr} = \mathsf{Var} \times \mathsf{Exp}$$
$$alloc(v, (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa})) = (v, e).$$

Polymorphic splitting: Pushdown poly/CFA Assuming we compiled the program from a programming language with let-bound polymorphism and marked which functions were let-bound, we can enable polymorphic splitting:

$$\widehat{Addr} = \mathsf{Var} + \mathsf{Var} \times \mathsf{Exp}$$
$$alloc(v, (\llbracket (f\ æ) \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\kappa})) = \begin{cases} (v, \llbracket (f\ æ) \rrbracket) & f \text{ is let-bound} \\ v & \text{otherwise.} \end{cases}$$

Pushdown k-CFA For pushdown k-CFA, we need to look beyond the current state and at the last k states. By concatenating the expressions in the last k states together, and pairing this sequence with a variable we get pushdown k-CFA:

$$\widehat{Addr} = \mathsf{Var} \times \mathsf{Exp}^k$$
$$\widehat{alloc}(v, \langle (e_1, \hat{\rho}_1, \hat{\sigma}_1, \hat{\kappa}_1), \ldots \rangle) = (v, \langle e_1, \ldots, e_k \rangle).$$

4.1 Partial orders

For each set $\hat{X}$ inside the abstract configuration-space, we use the natural partial order, $(\sqsubseteq_{\hat{X}}) \subseteq \hat{X} \times \hat{X}$. Abstract addresses and syntactic sets have flat partial orders. For the other sets, the partial order lifts:

- point-wise over environments:
$$\hat{\rho} \sqsubseteq \hat{\rho}' \text{ iff } \hat{\rho}(v) = \hat{\rho}'(v) \text{ for all } v \in dom(\hat{\rho});$$
- component-wise over closures:
$$(lam, \hat{\rho}) \sqsubseteq (lam, \hat{\rho}') \text{ iff } \hat{\rho} \sqsubseteq \hat{\rho}';$$
- point-wise over stores:
$$\hat{\sigma} \sqsubseteq \hat{\sigma}' \text{ iff } \hat{\sigma}(\hat{a}) \sqsubseteq \hat{\sigma}'(\hat{a}) \text{ for all } \hat{a} \in dom(\hat{\sigma});$$
- component-wise over frames:
$$(v, e, \hat{\rho}) \sqsubseteq (v, e, \hat{\rho}') \text{ iff } \hat{\rho} \sqsubseteq \hat{\rho}';$$
- element-wise over continuations:
$$\langle \hat{\phi}_1, \ldots, \hat{\phi}_n \rangle \sqsubseteq \langle \hat{\phi}'_1, \ldots, \hat{\phi}'_n \rangle \text{ iff } \hat{\phi}_i \sqsubseteq \hat{\phi}'_i; \text{ and}$$
- component-wise across configurations:
$$(e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \sqsubseteq (e, \hat{\rho}', \hat{\sigma}', \hat{\kappa}') \text{ iff } \hat{\rho} \sqsubseteq \hat{\rho}' \text{ and } \hat{\sigma} \sqsubseteq \hat{\sigma}' \text{ and } \hat{\kappa} \sqsubseteq \hat{\kappa}'.$$

4.2 Soundness

To prove soundness, an abstraction map α connects the concrete and abstract configuration-spaces:

$$\begin{aligned} \alpha(e, \rho, \sigma, \kappa) &= (e, \alpha(\rho), \alpha(\sigma), \alpha(\kappa)) \\ \alpha(\rho) &= \lambda v. \alpha(\rho(v)) \\ \alpha(\sigma) &= \lambda \hat{a}. \bigsqcup_{\alpha(a) = \hat{a}} \{\alpha(\sigma(a))\} \\ \alpha\langle \phi_1, \ldots, \phi_n \rangle &= \langle \alpha(\phi_1), \ldots, \alpha(\phi_n) \rangle \\ \alpha(v, e, \rho) &= (v, e, \alpha(\rho)) \\ &\alpha(a) \text{ is determined by the allocation functions.} \end{aligned}$$

It is then easy to prove that the abstract transition relation simulates the concrete transition relation:

Theorem 4.1. *If:*

$$\alpha(c) \sqsubseteq \hat{c} \textit{ and } c \Rightarrow c',$$

then there must exist $\hat{c}' \in \widehat{Conf}$ *such that:*

$$\alpha(c') \sqsubseteq \hat{c}' \textit{ and } \hat{c} \rightsquigarrow \hat{c}'.$$

Proof. The proof follows by case-wise analysis on the type of the expression in the configuration. It is a straightforward adaptation of similar proofs, such as that of [11] for k-CFA. ☐

5. The shift: From abstract CESK to rooted PDS

In the previous section, we constructed an infinite-state abstract interpretation of the CESK machine. The infinite-state nature of the abstraction makes it difficult to see how to answer static analysis questions. Consider, for instance, a control flow-question:

At the call site $(f\ æ)$, may a closure over lam be called?

If the abstracted CESK machine were a finite-state machine, an algorithm could answer this question by enumerating all reachable configurations and looking for an abstract configuration $(\llbracket (f\ æ) \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\kappa})$ in which $(lam, \_) \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$. However, because the abstracted CESK machine may contain an infinite number of reachable configurations, enumeration is not an option.

Fortunately, a shift in perspective reveals the abstracted CESK machine to be a rooted pushdown system. This shift permits the use of a control-state reachability algorithm in place of exhaustive search of the configuration-space. In this shift, a control-state is an expression-environment-store triple, and a stack character is a frame. Figure 5 defines the program-to-RPDS conversion function $\widehat{\mathcal{PDS}} : \mathsf{Exp} \to \mathbb{RPDS}$.

At this point, we can compute the root-reachable control states using a straightforward summarization algorithm [1, 17, 18]. This is the essence of CFA2.

6. Introspection for abstract garbage collection

Abstract garbage collection [14] yields large improvements in precision by using the abstract interpretation of garbage collection to make more efficient use of the finite address space available during analysis. Because of the way abstract garbage collection operates, it

$$\begin{aligned}
\widehat{\mathcal{PDS}}(e) &= (Q, \Gamma, \delta, q_0), \text{ where}\\
Q &= \mathsf{Exp} \times \widehat{Env} \times \widehat{Store}\\
\Gamma &= \widehat{Frame}\\
(q, \epsilon, q') \in \delta &\text{ iff } (q, \hat{\kappa}) \leadsto (q', \hat{\kappa}) \text{ for all } \hat{\kappa}\\
(q, \hat{\phi}_{-}, q') \in \delta &\text{ iff } (q, \hat{\phi} : \hat{\kappa}) \leadsto (q', \hat{\kappa}) \text{ for all } \hat{\kappa}\\
(q, \hat{\phi}'_{+}, q') \in \delta &\text{ iff } (q, \hat{\kappa}) \leadsto (q', \hat{\phi}' : \hat{\kappa}) \text{ for all } \hat{\kappa}\\
(q_0, \langle\rangle) &= \hat{\mathcal{I}}(e).
\end{aligned}$$

Figure 5. $\widehat{\mathcal{PDS}} : \mathsf{Exp} \to \mathbb{RPDS}$.

grants exact precision to the flow analysis of variables whose bindings die between invocations of the same abstract context. Because pushdown analysis grants exact precision in tracking return-flow, it is clearly advantageous to combine these techniques. Unfortunately, as we shall demonstrate, abstract garbage collection breaks the pushdown model by requiring full stack inspection to discover the root set.

Abstract garbage collection modifies the transition relation to conduct a "stop-and-copy" garbage collection before each transition. To do this, we define a garbage collection function $\hat{G} : \widehat{Conf} \to \widehat{Conf}$ on configurations:

$$\hat{G}(\overbrace{e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}}^{\hat{c}}) = (e, \hat{\rho}, \hat{\sigma}|Reachable(\hat{c}), \hat{\kappa}),$$

where the pipe operation $f|S$ yields the function f, but with inputs not in the set S mapped to bottom—the empty set. The reachability function $Reachable : \widehat{Conf} \to \mathcal{P}(\widehat{Addr})$ first computes the root set, and then the transitive closure of an address-to-address adjacency relation:

$$Reachable(\overbrace{e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}}^{\hat{c}}) = \left\{ \hat{a} : \hat{a}_0 \in Root(\hat{c}) \text{ and } \hat{a}_0 \overset{*}{\underset{\hat{\sigma}}{\rightharpoonup}} \hat{a} \right\},$$

where the function $Root : \widehat{Conf} \to \mathcal{P}(\widehat{Addr})$ finds the root addresses:

$$Root(e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) = range(\hat{\rho}) \cup StackRoot(\hat{\kappa}),$$

and the $StackRoot : \widehat{Kont} \to \mathcal{P}(\widehat{Addr})$ function finds roots down the stack:

$$StackRoot\langle(v_1, e_1, \hat{\rho}_1), \ldots, (v_n, e_n, \hat{\rho}_n)\rangle = \bigcup_i range(\hat{\rho}_i),$$

and the relation $(\rightharpoonup) \subseteq \widehat{Addr} \times \widehat{Store} \times \widehat{Addr}$ connects adjacent addresses:

$$\hat{a} \underset{\hat{\sigma}}{\rightharpoonup} \hat{a}' \text{ iff there exists } (lam, \hat{\rho}) \in \hat{\sigma}(\hat{a}) \text{ such that } \hat{a}' \in range(\hat{\rho}).$$

The new abstract transition relation is thus the composition of abstract garbage collection with the old transition relation:

$$(\leadsto_{\mathrm{GC}}) = (\leadsto) \circ \hat{G}$$

Problem: Stack traversal violates pushdown constraint In the formulation of pushdown systems, the transition relation is restricted to looking at the top frame, and even in less restricted formulations, at most a bounded number of frames can be inspected. Thus, the relation $(\leadsto_{\mathrm{GC}})$ cannot be computed as a straightforward pushdown analysis using summarization.

Solution: Introspective pushdown systems To accomodate the richer structure of the relation $(\leadsto_{\mathrm{GC}})$, we now define *introspective* pushdown systems. Once defined, we can embed the garbage-collecting abstract interpretation within this framework, and then focus on developing a control-state reachability algorithm for these systems.

An **introspective pushdown system** is a quadruple $M = (Q, \Gamma, \delta, q_0)$:

1. Q is a finite set of control states;
2. Γ is a stack alphabet;
3. $\delta \subseteq Q \times \Gamma^* \times \Gamma_{\pm} \times Q$ is a transition relation; and
4. q_0 is a distinguished root control state.

The second component in the transition relation is a realizable stack at the given control-state. This realizable stack distinguishes an introspective pushdown system from a general pushdown system. $\mathbb{IPDS}$ denotes the class of all introspective pushdown systems.

Determining how (or if) a control state q transitions to a control state q', requires knowing a path taken to the state q. Thus, we need to define reachability inductively. When $M = (Q, \Gamma, \delta, q_0)$, transition from the initial control state considers only empty stacks:

$$q_0 \underset{M}{\overset{g}{\longmapsto\!\!\!\!\rightarrow}} q \text{ iff } (q_0, \langle\rangle, g, q) \in \delta.$$

For non-root states, the paths to that state matter, since they determine the stacks realizable with that state:

$$q \underset{M}{\overset{g}{\longmapsto\!\!\!\!\rightarrow}} q' \text{ iff there exists } \vec{g} \text{ such that } q_0 \underset{M}{\overset{\vec{g}}{\longmapsto\!\!\!\!\rightarrow}} q \text{ and } (q, [\vec{g}], g, q') \in \delta,$$

$$\text{where } q \underset{M}{\overset{\langle g_1, \ldots, g_n \rangle}{\longmapsto\!\!\!\!\rightarrow}} q' \text{ iff } q \underset{M}{\overset{g_1}{\longmapsto\!\!\!\!\rightarrow}} q_1 \underset{M}{\overset{g_2}{\longmapsto\!\!\!\!\rightarrow}} \cdots \underset{M}{\overset{g_n}{\longmapsto\!\!\!\!\rightarrow}} q'.$$

6.1 Garbage collection in introspective pushdown systems

To convert the garbage-collecting, abstracted CESK machine into an introspective pushdown system, we use the function $\widehat{\mathcal{IPDS}} : \mathsf{Exp} \to \mathbb{IPDS}$:

$$\begin{aligned}
\widehat{\mathcal{IPDS}}(e) &= (Q, \Gamma, \delta, q_0)\\
Q &= \mathsf{Exp} \times \widehat{Env} \times \widehat{Store}\\
\Gamma &= \widehat{Frame}\\
(q, \hat{\kappa}, \epsilon, q') \in \delta &\text{ iff } \hat{G}(q, \hat{\kappa}) \leadsto (q', \hat{\kappa})\\
(q, \hat{\phi} : \hat{\kappa}, \hat{\phi}_{-}, q') \in \delta &\text{ iff } \hat{G}(q, \hat{\phi} : \hat{\kappa}) \leadsto (q', \hat{\kappa})\\
(q, \hat{\kappa}, \hat{\phi}_{+}, q') \in \delta &\text{ iff } \hat{G}(q, \hat{\kappa}) \leadsto (q', \hat{\phi} : \hat{\kappa})\\
(q_0, \langle\rangle) &= \hat{\mathcal{I}}(e).
\end{aligned}$$

7. Introspective reachability via Dyck state graphs

Having defined introspective pushdown systems and embedded our abstract, garbage-collecting semantics within them, we are ready to define control-state reachability for IDPSs.

We cast our reachability algorithm for introspective pushdown systems as finding a fixed-point, in which we incrementally accrete the reachable control states into a "Dyck state graph."

A **Dyck state graph** is a quadruple $G = (S, \Gamma, E, s_0)$, in which:

1. S is a finite set of nodes;
2. Γ is a set of frames;
3. $E \subseteq S \times \Gamma_\pm \times S$ is a set of stack-action edges; and
4. s_0 is an initial state;

such that for any node $s \in S$, it must be the case that:

$$(s_0, \langle\rangle) \underset{G}{\overset{*}{\longmapsto}} (s, \vec{\gamma}) \text{ for some stack } \vec{\gamma}.$$

In other words, a Dyck state graph is equivalent to a rooted pushdown system in which there is a legal path to every control state from the initial control state.[4] We use $\mathbb{DSG}$ to denote the class of Dyck state graphs. (Clearly, $\mathbb{DSG} \subset \mathbb{RPDS}$.)

Our goal is to compile an implicitly-defined introspective pushdown system into an explicited-constructed Dyck state graph. During this transformation, the per-state path considerations of an introspective pushdown are "baked into" the Dyck state graph. We can formalize this compilation process as a map, $\mathcal{DSG} : \mathbb{IPDS} \to \mathbb{DSG}$.

Given an introspective pushdown system $M = (Q, \Gamma, \delta, q_0)$, its equivalent Dyck state graph is $\mathcal{DSG}(M) = (S, \Gamma, E, q_0)$, where $s_0 = q_0$, the set S contains reachable nodes:

$$S = \left\{ q : q_0 \underset{M}{\overset{\vec{g}}{\longmapsto\!\!\!\!\rightarrow}} q \text{ for some stack-action sequence } \vec{g} \right\},$$

and the set E contains reachable edges:

$$E = \left\{ q \overset{g}{\rightarrowtail} q' : q \underset{M}{\overset{g}{\longmapsto\!\!\!\!\rightarrow}} q' \right\}.$$

Our goal is to find a method for computing a Dyck state graph from an introspective pushdown system.

7.1 Compiling to Dyck state graphs

We now turn our attention to compiling an introspective pushdown system (defined implicitly) into a Dyck state graph (defined explicitly). That is, we want an implementation of the function $\mathcal{DSG}$. To do so, we first phrase the Dyck state graph construction as the least fixed point of a monotonic function. This formulation provides a straightforward iterative method for computing the function $\mathcal{DSG}$.

The function $\mathcal{F} : \mathbb{IPDS} \to (\mathbb{DSG} \to \mathbb{DSG})$ generates the monotonic iteration function we need:

$$\begin{aligned}
\mathcal{F}(M) &= f, \text{ where} \\
M &= (Q, \Gamma, \delta, q_0) \\
f(S, \Gamma, E, s_0) &= (S', \Gamma, E', s_0), \text{ where} \\
S' &= S \cup \left\{ s' : s \in S \text{ and } s \underset{M}{\longmapsto\!\!\!\!\rightarrow} s' \right\} \cup \{s_0\} \\
E' &= E \cup \left\{ s \overset{g}{\rightarrowtail} s' : s \in S \text{ and } s \underset{M}{\overset{g}{\longmapsto\!\!\!\!\rightarrow}} s' \right\}.
\end{aligned}$$

Given an introspective pushdown system M, each application of the function $\mathcal{F}(M)$ accretes new edges at the frontier of the Dyck state graph.

[4] We chose the term *Dyck state graph* because the sequences of stack actions along valid paths through the graph correspond to substrings in Dyck languages. A **Dyck language** is a language of balanced, "colored" parentheses. In this case, each character in the stack alphabet is a color.

7.2 Computing a round of $\mathcal{F}$

The formalism obscures an important detail in the computation of an iteration: the transition relation ($\longmapsto\!\!\!\!\rightarrow$) for the introspective pushdown system must compute all possible stacks in determining whether or not there exists a transition. Fortunately, this is not as onerous as it seems: the set of all possible stacks for any given control-point is a regular language, and the finite automaton that encodes this language can be lifted (or read off) the structure of the Dyck state graph. The function $\mathit{Stacks} : \mathbb{DSG} \to S \to \mathbb{NFA}$ performs exactly this extraction:

$$\begin{aligned}
\mathit{Stacks}(\overbrace{S, \Gamma, E, s_0}^{M})(s) &= (S, \Gamma, \delta, s_0, \{s\}), \text{ where} \\
(s', \gamma, s'') \in \delta &\text{ if } (s', \gamma_+, s'') \in E \\
(s', \epsilon, s'') \in \delta &\text{ if } s' \underset{M}{\overset{\vec{g}}{\longmapsto\!\!\!\!\rightarrow}} s'' \text{ and } [\vec{g}] = \epsilon.
\end{aligned}$$

7.3 Correctness

Once the algorithm reaches a fixed point, the Dyck state graph is complete:

Theorem 7.1. $\mathcal{DSG}(M) = \text{lfp}(\mathcal{F}(M))$.

Proof. Let $M = (Q, \Gamma, \delta, q_0)$. Let $f = \mathcal{F}(M)$. Observe that $\text{lfp}(f) = f^n(\emptyset, \Gamma, \emptyset, q_0)$ for some n. When $N \subseteq M$, then it easy to show that $f(N) \subseteq M$. Hence, $\mathcal{DSG}(M) \supseteq \text{lfp}(\mathcal{F}(M))$.

To show $\mathcal{DSG}(M) \subseteq \text{lfp}(\mathcal{F}(M))$, suppose this is not the case. Then, there must be at least one edge in $\mathcal{DSG}(M)$ that is not in $\text{lfp}(\mathcal{F}(M))$. By the defintion of $\mathcal{DSG}(M)$, each edge must be part of a sequence of edges from the initial state. Let (s, g, s') be the first edge in its sequence from the initial state that is not in $\text{lfp}(\mathcal{F}(M)$. Because the proceeding edge is in $\text{lfp}(\mathcal{F}(M))$, the state s *is* in $\text{lfp}(\mathcal{F}(M))$. Let m be the lowest natural number such that s appears in $f^m(M)$. By the definition of f, this edge must appear in $f^{m+1}(M)$, which means it must also appear in $\text{lfp}(\mathcal{F}(M))$, which is a contradiction. Hence, $\mathcal{DSG}(M) \subseteq \text{lfp}(\mathcal{F}(M))$. ☐

7.4 Complexity

While decidability is the goal, it is straightforward to determine the complexity of this naïve fixed-point method. To determine the complexity of this algorithm, we ask two questions: how many times would the algorithm invoke the iteration function in the worst case, and how much does each invocation cost in the worst case? The size of the final Dyck state graph bounds the run-time of the algorithm. Suppose the final Dyck state graph has m states. In the worst case, the iteration function adds only a single edge each time. Between any two states, there is one ϵ-edge, one push edge, or some number of pop edges (at most $|\Gamma|$). Since there are at most $|\Gamma| m^2$ edges in the final graph, the maximum number of iterations is $|\Gamma| m^2$.

The cost of computing each iteration is harder to bound. The cost of determining whether to add a push edge is constant, as is the cost of adding an ϵ-edge. So the cost of determining all new push edges and new ϵ-edges to add is constant. Determining whether or not to add a pop edge is expensive. To add the pop edge $s \rightarrowtail^{\gamma-} s'$, we must prove that there exists a configuration-path to the control state s, in which the character γ is on the top of the stack. This reduces to a CFL-reachability query [9] at each node, the cost of which is $O(|\Gamma_\pm|^3 m^3)$ [8].

To summarize, in terms of the number of reachable control states, the complexity of this naive algorithm is:

$$O((|\Gamma| m^2) \times (|\Gamma_\pm|^3 m^3)) = O(|\Gamma|^4 m^5).$$

(As with summarization, it is possible to maintain a work-list and introduce an ϵ-closure graph to avoid spurious recomputation. This ultimately reduces complexity to $O(|\Gamma|^2 m^4)$.)

8. Implementation and evaluation

We have developed an implementation to produce the Dyck state graph of an introspective pushdown system. While the fixed-point computation 7.2 could be rendered directly as functional code, extending the classical summarization-based algorithm for pushdown reachability to introspective pushdown systems yields better performance. In this section we present a variant of such an algorithm and discuss results from an implementation that can analyze a large subset of the Scheme programming language.

8.1 Iterating over a DSG: An implementor's view

To synthesize a Dyck state graph from an introspective pushdown system, it is built incrementally—node by node, edge by edge. The naïve fixed point algorithm presented earlier, if implemented literally, would (in the worst case) have to re-examine the entire DSG to add each edge. To avoid such re-examination, our implementation adds ϵ-summary edges to the DSG.

In short, an ϵ-summary edge connects two control states if there exists a path between them with no net stack change—that is, all pushes are cancelled by corresponding pops. With ϵ-summary edges available, any change to the graph can be propagated directly to where it has an effect, and then any new ϵ-summary edges that propagation implies are added.

Whereas the correspondence between CESK and an IPDS is relatively straightforward, the relationship between a DSG and its original IPDS is complicated by the fact that the IPDS keeps track of the *whole* stack, whereas the DSG distributes (the same) stack information throughout its internal structure.

A classic reachability-based analysis for a pushdown system requires two mutually-dependent pieces of information in order to add another edge:

1. The topmost frame on a stack for a given control state q. This is essential for *return* transitions, as this frame should be popped from the stack and the store and the environment of a caller should be updated respectively.
2. Whether a given control state q is reachable or not from the initial state q_0 along realizable sequences of stack actions. For example, a path from q_0 to q along edges labeled "push, pop, pop, push" is not realizable: the stack is empty after the first pop, so the second pop cannot happen—let alone the subsequent push.

These two data are enough for a classic pushdown reachability summarization to proceed one step further. However, the presence of an abstract garbage collector, and the graduation to an *introspective* pushdown system, imposes the requirement for a third item of data:

3. For a given control state q, what are *all* possible frames that could happen to be *on* the stack at the moment the IPDS is in the state q?

It is possible to recompute these frames from scratch in each iteration using the NFA-extraction technique we described. But, it is easier to maintain per-node summaries, in the same spirit as ϵ-summary edges.

A version of the classic pushdown summarization algorithm that maintains the first two items is presented in [4], so we will just outline the key differences here.

The crux of the algorithm is to maintain for each node q' in the DSG, a set of *ϵ-predecessors*, i.e., nodes q, such that $q \longmapsto\!\!\!\!\twoheadrightarrow^{\vec{g}}_{M} q'$ and $[\vec{g}] = \epsilon$. In fact, only two out of three kinds of transitions can cause a change to the set of ϵ-predecessors for a particular node q: an addition of an ϵ-edge or a pop edge to the DSG.

It is easy to see why the second action might introduce new ϵ-paths and, therefore, new ϵ-predecessors. Consider, for example, adding the γ_--edge $q \rightarrowtail^{\gamma_-} q'$ into the following graph:

$$\begin{tikzcd} q_0 \arrow[r, "\gamma_+"] & q & q' \arrow[r, "\epsilon"] & q_1 \end{tikzcd}$$

As soon as this edge drops in, there becomes an "implicit" ϵ-edge between q_0 and q_1 because the net stack change between them is empty; the resulting graph looks like:

$$\begin{tikzcd} q_0 \arrow[r, "\gamma_+"] \arrow[rrr, bend left, dashed, "\epsilon"] & q \arrow[r, "\gamma_-"] & q' \arrow[r, "\epsilon"] & q_1 \end{tikzcd}$$

where we have illustrated the implicit ϵ-edge as a dashed line.

A little reflection on ϵ-predecessors and top frames reveals a mutual dependency between these items during the construction of a DSG. Informally:

- A *top frame* for a state q can be pushed as a direct predecessor, or as a direct predecessor to an ϵ-predecessor.
- When a new ϵ-edge $q \xrightarrow{\epsilon} q'$ is added, all ϵ-predecessors of q become also ϵ-predecessors of q'. That is, ϵ-summary edges are transitive.
- When a γ_--pop-edge $q \xrightarrow{\gamma_-} q'$ is added, new ϵ-predecessors of a state q_1 can be obtained by checking if q' is an ϵ-predecessor of q_1 and examining all existing ϵ-predecessors of q, such that γ_+ is their possible top frame: this situation is similar to the one depicted in the example above.

The third component—*all* possible frames on the stack for a state q—is straightforward to compute with ϵ-predecessors: starting from q, trace out only the edges which are labeled ϵ (summary or otherwise) or γ_+. The frame for any action γ_+ in this trace is a possible stack action. Since these sets grow monotonically, it is easy to cache the results of the trace, and in fact, propagate incremental changes to these caches when new ϵ-summary or γ_+ nodes are introduced. Our implementation directly reflects the optimizations discussed above.

8.2 Experimental results

A fair comparison between different families of analyses should compare both precision and speed. We have extended an existing

Program	Exp	Var	k	k-CFA			k-PDCFA			k-CFA + GC			k-PDCFA + GC		
mj09	19	8	0	83	107	4	38	38	4	36	39	4	33	32	4
			1	454	812	1	44	48	1	34	35	1	32	31	1
eta	21	13	0	63	74	4	34	34	6	28	27	8	28	27	8
			1	33	33	8	32	31	8	28	27	8	28	27	8
kcfa2	20	10	0	194	236	3	36	35	4	35	43	4	35	34	4
			1	970	1935	1	87	144	2	35	34	2	35	34	2
kcfa3	25	13	0	272	327	4	58	63	5	53	52	5	53	52	5
			1	> 7119	> 14201	≤ 1	1761	4046	2	53	52	2	53	52	2
blur	40	20	0	> 1419	> 2435	≤ 3	280	414	3	274	298	9	164	182	9
			1	261	340	9	177	189	9	169	189	9	167	182	9
loop2	41	14	0	228	252	4	113	122	4	86	93	4	70	74	4
			1	> 10867	> 16040	≤ 3	411	525	3	151	163	3	145	156	3
sat	63	31	0	> 5362	> 7610	≤ 6	775	979	6	1190	1567	6	321	384	6
			1	> 8395	> 12391	≤ 6	7979	10299	6	982	1330	7	107	106	13

Figure 6. Benchmark results. The first three columns provide the name of a benchmark, the number of expressions and variables in the program in the ANF, respectively. For each of eight combinations of pushdown analysis, $k \in \{0, 1\}$ and garbage collection on or off, the first two columns in a group show the number of *control states* and transitions/DSG edges computed during the analysis (for both less is better). The third column presents the amount of *singleton* variables, i.e, how many variables have a single lambda flow to them (more is better). Inequalities for some results denote the case when the analysis did not finish within 30 minutes. For such cases we can only report an upper bound of singleton variables as this number can only decrease.

implementation of k-CFA to optionally enable pushdown analysis, abstract garbage collection or both. Our implementation source and benchmarks are available:

```
http://github.com/ilyasergey/reachability
```

As expected, the fused analysis does at least as well as the best of either analysis alone in terms of singleton flow sets (a good metric for program optimizability) and better than both in some cases. Also worthy of note is the dramatic reduction in the size of the abstract transition graph for the fused analysis—even on top of the already large reductions achieved by abstract gabarge collection and pushdown flow analysis individually. The size of the abstract transition graph is a good heuristic measure of the temporal reasoning ability of the analysis, *e.g.*, its ability to support model-checking of safety and liveness properties [12].

In order to exercise both well-known and newly-presented instances of CESK-based CFAs, we took a series of small benchmarks exhibiting archetypal control-flow patterns (see Figure 6). Most benchmarks are taken from the CFA literature: `mj09` is a running example from the work of Midtgaard and Jensen designed to exhibit a non-trivial return-flow behavior, `eta` and `blur` test common functional idioms, mixing closures and eta-expansion, `kcfa2` and `kcfa3` are two worst-case examples extracted from Van Horn and Mairson's proof of k-CFA complexity [21], `loop2` is an example from the Might's dissertation that was used to demonstrate the impact of abstract GC [11, Section 13.3], `sat` is a brute-force SAT-solver with backtracking.

8.2.1 Comparing precision

In terms of precision, the fusion of pushdown analysis and abstract garbage collection substantially cuts abstract transition graph sizes over one technique alone.

We also measure singleton flow sets as a heuristic metric for precision. Singleton flow sets are a necessary precursor to optimizations such as flow-driven inlining, type-check elimination and constant propagation. Here again, the fused analysis prevails as the best-of- or better-than-both-worlds.

Program	0-CFA		0-PDCFA		1-CFA		1-PDCFA	
mj09	$1''$	ϵ	ϵ	$1''$	$4''$	ϵ	ϵ	ϵ
eta	ϵ	ϵ	ϵ	ϵ	$1''$	ϵ	ϵ	ϵ
kcfa2	$1''$	ϵ	ϵ	$1''$	$24''$	ϵ	$1''$	ϵ
kcfa3	$2''$	ϵ	ϵ	$1''$	∞	$1''$	$58''$	$2''$
blur	∞	$7''$	$2'$	$50''$	$4'$	$30''$	$11''$	$55''$
loop2	$36''$	$1''$	$29''$	$16''$	∞	$5''$	$13'$	$2'$
sat	∞	$45''$	$6'$	$19'$	∞	$3'$	$12'$	$37''$

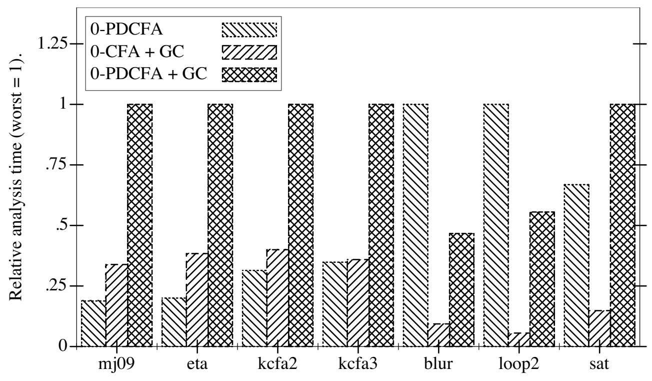
Figure 7. We ran our benchmark suite on a 2 Core 2.66 GHz OS X machine with 4 Gb RAM. For each of the four analyses the left column denotes the values obtained with no abstract collection, and the right one—with GC on. The results of the analyses are presented in minutes ($'$) or seconds ($''$), where ϵ means a value less than 1 second and ∞ stands for an analysis, which has been interrupted due to the an execution time greater than 30 minutes.

Running on the benchmarks, we have revalidated hypotheses about the improvements to precision granted by both pushdown analysis [22] and abstract garbage collection [11]. The table in Figure 6 contains our detailed results on the precision of the analysis.

8.2.2 Comparing speed

In the original work on CFA2, Vardoulakis and Shivers present experimental results with a remark that the running time of the analysis is proportional to the size of the reachable states [22, Section 6]. There is a similar correlation in the fused analysis, but it is not as strong or as absolute. From examination of the results, this appears to be because small graphs can have large stores inside each state, which increases the cost of garbage collection (and thus transition) on a per-state basis, and there is some additional per-transition overhead involved in maintaining the caches inside the Dyck state graph. Table 7 collects absolute execution times for comparison.

It follows from the results that pure machine-style k-CFA is always significantly worse in terms of execution time than either with GC or push-down system. The histogram on Figure 8 presents



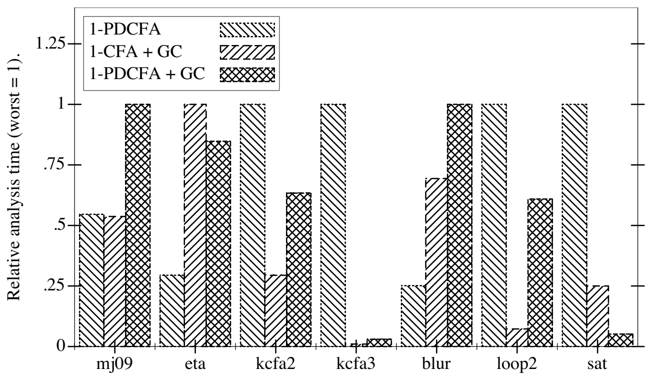


Figure 8. Analysis times relative to worst (= 1) in class; smaller is better. On the left is the monovariant 0CFA class of analyses, on the right is the polyvariant 1CFA class of analyses. (Non-GC k-CFA omitted.)

normalized relative times of analyses' executions. About half the time, the fused analysis is faster than one of pushdown analysis or abstract garbage collection. And about a tenth of the time, it is faster than both.[5] When the fused analysis is slower than both, it is generally not much worse than twice as slow as the next slowest analysis.

Given the already substantial reductions in analysis times provided by collection and pushdown anlysis, the amortized penalty is a small and acceptable price to pay for improvements to precision.

9. Related work

Garbage-collecting pushdown control-flow analysis draws on work in higher-order control-flow analysis [19], abstract machines [5] and abstract interpretation [3].

Context-free analysis of higher-order programs The motivating work for our own is Vardoulakis and Shivers very recent discovery of CFA2 [22]. CFA2 is a table-driven summarization algorithm that exploits the balanced nature of calls and returns to improve return-flow precision in a control-flow analysis. Though CFA2 exploits context-free languages, context-free languages are not explicit in its formulation in the same way that pushdown systems are explicit in our presentation of pushdown flow analysis. With respect to CFA2, our pushdown flow analysis is also polyvariant/context-sensitive (whereas CFA2 is monovariant/context-insensitive), and it covers direct-style.

On the other hand, CFA2 distinguishes stack-allocated and store-allocated variable bindings, whereas our formulation of pushdown control-flow analysis does not: it allocates all bindings in the store. If CFA2 determines a binding can be allocated on the stack, that binding will enjoy added precision during the analysis and is not subject to merging like store-allocated bindings. While we could incorporate such a feature in our formulation, it is not necessary for achieving "pushdownness," and in fact, it could be added to classical finite-state CFAs as well.

[5] The SAT-solving bechmark showed a dramatic improvement with the addition of context-sensitivity. Evaluation of the results showed that context-sensitivity provided enough fuel to eliminate most of the non-determinism from the analysis.

Calculation approach to abstract interpretation Midtgaard and Jensen [10] systematically calculate 0CFA using the Cousot-Cousot-style calculational approach to abstract interpretation [2] applied to an ANF λ-calculus. Like the present work, Midtgaard and Jensen start with the CESK machine of Flanagan *et al.* [6] and employ a reachable-states model.

The analysis is then constructed by composing well-known Galois connections to reveal a 0CFA incorporating reachability. The abstract semantics approximate the control stack component of the machine by its top element. The authors remark monomorphism materializes in two mappings: "one mapping all bindings to the same variable," the other "merging all calling contexts of the same function." Essentially, the pushdown 0CFA of Section 4 corresponds to Midtgaard and Jensen's analysis when the latter mapping is omitted and the stack component of the machine is not abstracted.

CFL- and pushdown-reachability techniques This work also draws on CFL- and pushdown-reachability analysis [1, 8, 17, 18]. For instance, ϵ-closure graphs, or equivalent variants thereof, appear in many context-free-language and pushdown reachability algorithms. For our analysis, we implicitly invoked these methods as subroutines. When we found these algorithms lacking (as with their enumeration of control states), we developed Dyck state graph construction.

CFL-reachability techniques have also been used to compute classical finite-state abstraction CFAs [9] and type-based polymorphic control-flow analysis [16]. These analyses should not be confused with pushdown control-flow analysis, which is computing a fundamentally more precise kind of CFA. Moreover, Rehof and Fahndrich's method is cubic in the size of the *typed* program, but the types may be exponential in the size of the program. Finally, our technique is not restricted to typed programs.

Model-checking higher-order recursion schemes There is terminology overlap with work by Kobayashi [7] on model-checking higher-order programs with higher-order recursion schemes, which are a generalization of context-free grammars in which productions can take higher-order arguments, so that an order-0 scheme is a context-free grammar. Kobyashi exploits a result by Ong [15] which shows that model-checking these recursion schemes is decidable (but ELEMENTARY-complete) by transforming higher-order programs into higher-order recursion schemes.

Given the generality of model-checking, Kobayashi's technique may be considered an alternate paradigm for the analysis of higher-order programs. For the case of order-0, both Kobayashi's technique and our own involve context-free languages, though ours is for control-flow analysis and his is for model-checking with respect to a temporal logic. After these surface similarities, the techniques diverge. In particular, higher-order recursions schemes are limited to model-checking programs in the simply-typed lambda-calculus with recursion.

10. Conclusion

Our motivation was to further probe the limits of decidability for pushdown flow analysis of higher-order programs by enriching it with abstract garbage collection. We found that abstract garbage collection broke the pushdown model, but not irreparably so. By casting abstract garbage collection in terms of an introspective pushdown system and synthesizing a new control-state reachability algorithm, we have demonstrated the decidability of fusing two powerful analytic techniques.

As a byproduct of our formulation, it was also easy to demonstrate how polyvariant/context-sensitive flow analyses generalize to a pushdown formulation, and we lifted the need to transform to continuation-passing style in order to perform pushdown analysis.

Our empirical evaluation is highly encouraging: it shows that the fused analysis provides further large reductions in the size of the abstract transition graph—a key metric for interprocedural control-flow precision. And, in terms of singleton flow sets—a heuristic metric for optimizability—the fused analysis proves to be a "better-than-both-worlds" combination.

Thus, we provide a sound, precise and polyvariant introspective pushdown analysis for higher-order programs.

Acknowledgments

We thank our anonymous reviewers for their detailed comments on the submitted paper. This material is based on research sponsored by DARPA under the programs Automated Program Analysis for Cybersecurity (FA8750-12-2-0106) and Clean-Slate Resilient Adaptive Hosts (CRASH). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] BOUAJJANI, A., ESPARZA, J., AND MALER, O. Reachability analysis of pushdown automata: Application to Model-Checking. In *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory* (1997), Springer-Verlag, pp. 135–150.

[2] COUSOT, P. The calculational design of a generic abstract interpreter. In *Calculational System Design*, M. Broy and R. Steinbrüggen, Eds. 1999.

[3] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages* (1977), ACM Press, pp. 238–252.

[4] EARL, C., MIGHT, M., AND VAN HORN, D. Pushdown control-flow analysis of higher-order programs. In *Proceedings of the 2010 Workshop on Scheme and Functional Programming* (Aug. 2010).

[5] FELLEISEN, M., AND FRIEDMAN, D. P. A calculus for assignments in higher-order languages. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1987), ACM, pp. 314+.

[6] FLANAGAN, C., SABRY, A., DUBA, B. F., AND FELLEISEN, M. The essence of compiling with continuations. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (June 1993), ACM, pp. 237–247.

[7] KOBAYASHI, N. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL '09: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2009), POPL '09, ACM, pp. 416–428.

[8] KODUMAL, J., AND AIKEN, A. The set constraint/CFL reachability connection in practice. *SIGPLAN Not. 39* (June 2004), 207–218.

[9] MELSKI, D., AND REPS, T. W. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science 248*, 1-2 (Oct. 2000), 29–98.

[10] MIDTGAARD, J., AND JENSEN, T. P. Control-flow analysis of function calls and returns by abstract interpretation. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (2009), pp. 287–298.

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

[12] MIGHT, M., CHAMBERS, B., AND SHIVERS, O. Model checking via Gamma-CFA. In *Verification, Model Checking, and Abstract Interpretation* (Jan. 2007), pp. 59–73.

[13] MIGHT, M., AND SHIVERS, O. Environment analysis via Delta-CFA. In *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2006), ACM, pp. 127–140.

[14] MIGHT, M., AND SHIVERS, O. Improving flow analyses via Gamma-CFA: Abstract garbage collection and counting. In *ICFP '06: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming* (2006), ACM, pp. 13–25.

[15] ONG, C. H. L. On Model-Checking trees generated by Higher-Order recursion schemes. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)* (2006), pp. 81–90.

[16] REHOF, J., AND FÄHNDRICH, M. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2001), ACM, pp. 54–66.

[17] REPS, T. Program analysis via graph reachability. *Information and Software Technology 40*, 11-12 (Dec. 1998), 701–726.

[18] REPS, T., SCHWOON, S., JHA, S., AND MELSKI, D. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming 58*, 1-2 (2005), 206–263.

[19] SHIVERS, O. G. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[20] SIPSER, M. *Introduction to the Theory of Computation*, 2 ed. Course Technology, Feb. 2005.

[21] VAN HORN, D., AND MAIRSON, H. G. Deciding *k*CFA is complete for EXPTIME. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming* (2008), pp. 275–282.

[22] VARDOULAKIS, D., AND SHIVERS, O. Cfa2: a Context-Free Approach to Control-Flow Analysis. In *European Symposium on Programming (ESOP)* (2010), vol. 6012 of *LNCS*, pp. 570–589.

[23] WRIGHT, A. K., AND JAGANNATHAN, S. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems 20*, 1 (Jan. 1998), 166–207.