Pushdown Control-Flow Analysis of Higher-Order Programs

Christopher Earl Matthew Might
University of Utah
{cwearl,might}@cs.utah.edu

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

Abstract

Context-free approaches to static analysis gain precision over classical approaches by perfectly matching returns to call sites—a property that eliminates spurious interprocedural paths. Vardoulakis and Shivers's recent formulation of CFA2 showed that it is possible (if expensive) to apply context-free methods to higher-order languages and gain the same boost in precision achieved over first-order programs.

To this young body of work on context-free analysis of higher-order programs, we contribute a pushdown control-flow analysis framework, which we derive as an abstract interpretation of a CESK machine with an unbounded stack. One instantiation of this framework marks the first polyvariant pushdown analysis of higher-order programs; another marks the first polynomial-time analysis. In the end, we arrive at a framework for control-flow analysis that can efficiently compute pushdown generalizations of classical control-flow analyses.

1. Introduction

Static analysis is bound by theory to accept diminished precision as the price of decidability. The easiest way to guarantee decidability for a static analysis is to restrict it to a finite state-space. Not surprisingly, finite state-spaces have become a crutch.

Whenever an abstraction maps an infinite (concrete) state-space down to the finite state-space of a static analysis, the pigeon-hole principle forces merging. Distinct execution paths and values can and do become indistinguishable under abstraction, *e.g.*, 3 and 4 both abstract to the same value: **positive**.

Our message is that finite abstraction goes too far: we can abstract into an infinite state-space to improve precision, yet remain decidable. Specifically, we can abstract the concrete semantics of a higher-order language into a pushdown automaton (PDA). As an infinite-state system, a PDA-based abstraction preserves more information than a classical finite-state analysis. Yet, being less powerful than a Turing machine, properties important for computing control-flow analysis (e.g. emptiness, intersection with regular languages, reachability) remain decidable.

1.1 The problem with merging

A short example provides a sense of how the inevitable merging that occurs under a finite abstraction harms precision. Shivers's 0CFA [Shivers 1991] produces spurious data-flows and return-flows in the following example:

```
(let* ((id (lambda (x) x))
       (a  (id 3))
       (b  (id 4)))
  a)
```

0CFA says that the flow set for the variable `a` contains both 3 and 4. In fact, so does the flow set for the variable `b`. For return-flow,[1] 0CFA says that the invocation of `(id 4)` may return to the invocation of `(id 3)` or `(id 4)` and vice versa; that is, according to Shivers's 0CFA, this program contains a loop.

To combat merging, control-flow analyses have focused on increasing context-sensitivity [Shivers 1991]. Context-sensitivity tries to qualify any answer that comes back from a CFA with a context in which that answer is valid. That is, instead of answering "λ_{42} may flow to variable v_{13}," a context-sensitive analysis might answer "λ_{42} may flow to variable v_{13} *when bound after calling* f." While context-sensitivity recovers some lost precision, it is no silver bullet. A finite-state analysis stores only a finite amount of program context to discriminate data- and control-flows during analysis. Yet, the pigeon-hole principle remains merciless: as long as the state-space is finite, merging is inevitable for some programs.

Of all the forms of merging, the most pernicious is the merging of return-flow information. As the example shows, a finite-state control-flow analysis will lose track of where return-points return once the maximum bound on context is exceeded. *Even in programs with no higher-order functions*, return-flow merging will still happen during control-flow analysis.

1.2 A first shot: CFA2

Vardoulakis and Shivers's recent work on CFA2 [Vardoulakis and Shivers 2010] constitutes an opening salvo on ending the return-flow problem for the static analysis of higher-order programs. CFA2 employs an implicit pushdown system that models the stack of a program. CFA2 solves the return-flow problem for higher-order programs, but it has drawbacks:

1. CFA2 allows only monovariant precision.
2. CFA2 has exponential complexity in the size of the program.
3. CFA2 is restricted to continuation-passing style.

Our solution overcomes all three drawbacks: it allows polyvariant precision, we can widen it to $O(n^6)$-time complexity in the monovariant case and we can operate on direct-style programs.

1.3 Our solution: Abstraction to pushdown systems

To prevent return-flow merging during higher-order control-flow analysis, we abstract into an explicit pushdown system instead of a finite-state machine. The program stack, which determines return-flow, will remain unbounded and become the pushdown stack. As a result, return-flow information will never be merged: in the abstract semantics, a function returns only whence it was called.

* Supported by the National Science Foundation under grant 0937060 to the Computing Research Association for the CIFellow Project.

[1] "Return-flow" analysis asks to which call sites a given return point may return. In the presence of tail calls, which break the balance between calls and returns, return-flow analysis differs from control-flow analysis.

1.4 Overview

This paper is organized as follows: first, we define a variant of the CESK machine [Felleisen and Friedman 1987] for the A-Normal Form λ-calculus [Flanagan et al. 1993]. In performing analysis, we wish to soundly approximate intensional properties of this machine when it evaluates a given program. To do so, we construct an abstract interpretation of the machine. The abstracted CESK machine operates much like its concrete counterpart and soundly approximates its behavior, but crucially, many properties of the concrete machine that are undecidable become decidable when considered against the abstracted machine (e.g. "is a given machine configuration reachable?" becomes a decidable property).

The abstract counterpart to the CESK machine is constructed by bounding the store component of the machine to some finite size. However, the stack component (represented as a continuation) is left unabstracted. (This is in contrast to finite-state abstractions that store-allocate continuations [Van Horn and Might 2010].) Unlike most higher-order abstract interpreters, the unbounded stack implies this machine has a potentially infinite set of reachable machine configurations, and therefore enumerating them is not a feasible approach to performing analysis.

Instead, we demonstrate how properties can be decided by transforming the abstracted CESK machine into an equivalent pushdown automaton. We then reduce higher-order control-flow analysis to deciding the non-emptiness of a language derived from the PDA. (This language happens to be the intersection of a regular language and the context-free language described by the PDA.) This approach—though concise, precise and decidable—is formidably expensive, with complexity doubly exponential in the size of the program.

We simplify the algorithm to merely exponential in the size of the input program by reducing the control-flow problem to pushdown reachability [Bouajjani et al. 1997]. Unfortunately, the abstracted CESK machine has an exponential number of control states with respect to the size of the program. Thus, pushdown reachability for higher-order programs appears to be inherently exponential.

Noting that most control states in the abstracted CESK machine are actually unreachable, we present a fixed-point algorithm for deciding pushdown reachability that is polynomial-time in the number of *reachable* control states. Since the pushdown systems produced by abstracted CESK machines are sparse, such algorithms, though exponential in the worst case, are reasonable options. Yet, we can do better.

Next, we add an ϵ-closure graph (a graph encoding no-stack-change reachability) and a work-list to the fixed-point algorithm. Together, these lower the cost of finding the reachable states of a pushdown system from $O(|\Gamma|^4 m^5)$ to $O(|\Gamma|^2 m^4)$, where Γ is the stack alphabet and m is the number of reachable control states.

To drop the complexity of our analysis to polynomial-time in the size of the input program, we must resort to both widening and monovariance. Widening with a single-threaded store and using a monovariant allocation strategy yields a pushdown control-flow analysis with polynomial-time complexity, at $O(n^6)$, where n is the size of the program.

Finally, we briefly highlight applications of pushdown control-flow analyses that are outside the reach of classical ones, discuss related work, and conclude.

2. Pushdown preliminaries

In this work, we make use of both pushdown systems and pushdown automata. (A pushdown automaton is a specific kind of pushdown system.) There are many (equivalent) definitions of these machines in the literature, so we adapt our own definitions from [Sipser 2005]. Even those familiar with pushdown theory may want to skim this section to pick up our notation.

2.1 Syntactic sugar

When a triple (x, ℓ, x') is an edge in a labeled graph, a little syntactic sugar aids presentation:

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

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. Pushdown systems do much stack manipulation, so to represent this more concisely, we 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_+ and a pop character γ_-; it also contains a no-stack-change indicator, ϵ:

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

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

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.

We use $\mathbb{PDS}$ to denote the class of all pushdown systems.

Unlike the more widely known pushdown automaton, a pushdown system *does not recognize a language*.

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

- The **configurations** of this machine—$\mathit{Configs}(M)$—are pairs over control states and stacks:

$$\mathit{Configs}(M) = Q \times \Gamma^*.$$

- The labeled **transition relation** $(\longmapsto_M) \subseteq \mathit{Configs}(M) \times \Gamma_\pm \times \mathit{Configs}(M)$ determines whether one configuration may transition to another while performing the given stack action:

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

- 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 $c_0, \ldots, c_n$.

- For the transitive closure:

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

Note Some texts define the transition relation δ so that $\delta \subseteq Q \times \Gamma \times Q \times \Gamma^*$. In these texts, $(q, \gamma, q', \vec{\gamma}) \in \delta$ means, "if in control state q while the character γ is on top, pop the stack, transition to control state q' and push $\vec{\gamma}$." Clearly, we can convert between these two representations by introducing extra control states to our representation when it needs to push multiple characters.

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 a the **root-reachable transition relation**:

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

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 as well:

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

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:

$$q \underset{M}{\mapstochar\twoheadrightarrow} q' \text{ iff } q \underset{M}{\overset{g}{\mapstochar\twoheadrightarrow}} q' \text{ for some action } g.$$

2.5 Pushdown automata

A **pushdown automaton** is a generalization of a rooted pushdown system, a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F, \vec{\gamma})$ in which:

1. Σ is an input alphabet;
2. $\delta \subseteq Q \times \Gamma_{\pm} \times (\Sigma \cup \{\epsilon\}) \times Q$ is a transition relation;
3. $F \subseteq Q$ is a set of accepting states; and
4. $\vec{\gamma} \in \Gamma^*$ is the initial stack.

We use $\mathbb{PDA}$ to denote the class of all pushdown automata.

Pushdown automata recognize languages over their input alphabet. To do so, their transition relation may optionally consume an input character upon transition. Formally, a PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F, \vec{\gamma})$ recognizes the language $\mathcal{L}(M) \subseteq \Sigma^*$:

$$\begin{aligned}
&\epsilon \in \mathcal{L}(M) \text{ if } q_0 \in F \\
&aw \in \mathcal{L}(M) \text{ if } \delta(q_0, \gamma_+, a, q') \text{ and } w \in \mathcal{L}(Q, \Sigma, \Gamma, \delta, q', F, \gamma : \vec{\gamma}) \\
&aw \in \mathcal{L}(M) \text{ if } \delta(q_0, \epsilon, a, q') \text{ and } w \in \mathcal{L}(Q, \Sigma, \Gamma, \delta, q', F, \vec{\gamma}) \\
&aw \in \mathcal{L}(M) \text{ if } \delta(q_0, \gamma_-, a, q') \text{ and } w \in \mathcal{L}(Q, \Sigma, \Gamma, \delta, q', F, \vec{\gamma}') \\
&\qquad\qquad\qquad \text{where } \vec{\gamma} = \langle \gamma, \gamma_2, \ldots, \gamma_n \rangle \text{ and } \vec{\gamma}' = \langle \gamma_2, \ldots, \gamma_n \rangle,
\end{aligned}$$

where a is either the empty string ϵ or a single character.

3. Setting: A-Normal Form λ-calculus

Since our goal is to create pushdown control-flow analyses of *higher-order languages*, we choose to operate on the λ-calculus. For simplicity of the concrete and abstract semantics, we choose to analyze programs in A-Normal Form, however this is strictly a cosmetic choice; all of our results can be replayed *mutatis mutandis* in a direct-style setting. ANF enforces an order of evaluation and it requires that all arguments to a function be atomic:

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

We use the CESK machine of Felleisen and Friedman [1987] to specify the semantics of ANF. We have chosen the CESK machine because it has an explicit stack, and under abstraction, the stack component of our CESK machine will become the stack component of a pushdown system.

First, we define a set of configurations (Conf) for this machine:

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

To define the semantics, we need five items:

1. $\mathcal{I} : \mathsf{Exp} \rightarrow \mathit{Conf}$ injects an expression into a configuration.
2. $\mathcal{A} : \mathsf{Atom} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Clo}$ evaluates atomic expressions.
3. $\mathcal{E} : \mathsf{Exp} \rightarrow \mathcal{P}(\mathit{Conf})$ computes the set of reachable machine configurations for a given program.
4. $(\Rightarrow) \subseteq \mathit{Conf} \times \mathit{Conf}$ transitions between configurations.
5. $\mathit{alloc} : \mathsf{Var} \times \mathit{Conf} \rightarrow \mathit{Addr}$ chooses fresh store addresses for newly bound variables.

Program injection The program injection function pairs an expression with an empty environment, an empty store and an empty stack to create the initial configuration:

$$c_0 = \mathcal{I}(e) = (e, [], [], \langle\rangle).$$

Atomic expression evaluation The atomic expression evaluator, $\mathcal{A} : \mathsf{Atom} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Clo}$, returns the value of an atomic expression in the context of an environment and a store:

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

Reachable configurations The evaluator $\mathcal{E} : \mathsf{Exp} \rightarrow \mathcal{P}(\mathit{Conf})$ returns all configurations reachable from the initial configuration:

$$\mathcal{E}(e) = \{c : \mathcal{I}(e) \Rightarrow^* c\}\,.$$

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

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

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

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

Function return pops a stack frame:

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

Allocation The address-allocation function is an opaque parameter in this semantics. We have done this so that the forthcoming abstract semantics may also parameterize allocation, and in so doing provide a 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:

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

4. An infinite-state abstract interpretation

Our goal is to statically bound the higher-order control-flow of the CESK machine of the previous section. So, we are going to conduct an abstract interpretation.

Since we are concerned with return-flow precision, we are going to abstract away less information than we normally would. Specifically, we are going to construct an infinite-state abstract interpretation of the CESK machine by leaving its stack unabstracted. (With an infinite-state abstraction, the usual approach for computing the static analysis—exploring the abstract configurations reachable from some initial configuration—simply will not work. Subsequent sections focus on finding an algorithm that can compute a finite representation of the reachable abstract configurations of the abstracted CESK machine.)

For the abstract interpretation of the CESK machine, we need an abstract configuration-space (Figure 1). To construct one, 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. To construct the abstract transition relation, we need five components analogous to those from the concrete semantics.

Program injection The abstract injection function $\hat{\mathcal{I}} : \mathsf{Exp} \to \widehat{\mathit{Conf}}$ 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, $\hat{\mathcal{A}} : \mathsf{Atom} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \to \mathcal{P}(\widehat{\mathit{Clo}})$, returns the value of an atomic expression in the context of an environment and a store; note how it returns a set:

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

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

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

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

Figure 1. The abstract configuration-space.

Because there are an infinite number of abstract configurations, a naïve implementation of this function may not terminate. In Sections 5 through 8, we show that there is a way to compute a finite representation of this set.

Transition relation The abstract transition relation $(\leadsto) \subseteq \widehat{\mathit{Conf}} \times \widehat{\mathit{Conf}}$ 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:

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

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:

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

A function return pops a stack frame:

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

Allocation, polyvariance and context-sensitivity In the abstract semantics, the abstract allocation function $\widehat{\mathit{alloc}} : \mathsf{Var} \times \widehat{\mathit{Conf}} \to \widehat{\mathit{Addr}}$ determines the polyvariance of the analysis (and, by extension, its context-sensitivity). In a control-flow analysis, *polyvariance* literally refers to the number of abstract addresses (variants) there are for each variable. By selecting the right abstract allocation function, we can instantiate pushdown versions of classical flow analyses.

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

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

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

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

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:

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

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:

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

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:
$$\hat{\rho} \sqsubseteq \hat{\rho}' \text{ iff } \hat{\rho}(v) = \hat{\rho}'(v) \text{ for all } v \in \mathit{dom}(\hat{\rho}).$$
- The partial orders lifts component-wise over closures:
$$(\mathit{lam}, \hat{\rho}) \sqsubseteq (\mathit{lam}, \hat{\rho}') \text{ iff } \hat{\rho} \sqsubseteq \hat{\rho}'.$$
- The partial order lifts point-wise over stores:
$$\hat{\sigma} \sqsubseteq \hat{\sigma}' \text{ iff } \hat{\sigma}(\hat{a}) \sqsubseteq \hat{\sigma}'(\hat{a}) \text{ for all } \hat{a} \in \mathit{dom}(\hat{\sigma}).$$
- The partial order lifts component-wise over frames:
$$(v, e, \hat{\rho}) \sqsubseteq (v, e, \hat{\rho}') \text{ iff } \hat{\rho} \sqsubseteq \hat{\rho}'.$$
- The partial order lifts element-wise over continuations:
$$\langle \hat{\phi}_1, \ldots, \hat{\phi}_n \rangle \sqsubseteq \langle \hat{\phi}'_1, \ldots, \hat{\phi}'_n \rangle \text{ iff } \hat{\phi}_i \sqsubseteq \hat{\phi}'_i.$$
- The partial order lifts 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, we need an abstraction map α that connects the concrete and abstract configuration-spaces:

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

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

Figure 2. $\widehat{\mathcal{PDA}} : \mathsf{Exp} \to \mathbb{PDA}$.

It is 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 $\hat{c}' \in \widehat{\mathit{Conf}}$ *such that:*

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

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 Might [2007] for k-CFA. □

5. From the abstracted CESK machine to a PDA

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 $(\mathit{lam}, \_) \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$. However, because the abstracted CESK machine may contain an infinite number of reachable configurations, an algorithm cannot enumerate them.

Fortunately, we can recast the abstracted CESK as a special kind of infinite-state system: a pushdown automaton (PDA). Pushdown automata occupy a sweet spot in the theory of computation: they have an infinite configuration-space, yet many useful properties (*e.g.* word membership, non-emptiness, control-state reachability) remain decidable. Once the abstracted CESK machine becomes a PDA, we can answer the control-flow question by checking whether a specific regular language, when intersected with the language of the PDA, turns into the empty language.

The recasting as a PDA is a shift in perspective. A configuration has an expression, an environment and a store. A stack character is a frame. We choose to make the alphabet the set of control states, so that the language accepted by the PDA will be sequences of control-states visited by the abstracted CESK machine. Thus, every transition will consume the control-state to which it transitioned as an input character. Figure 2 defines the program-to-PDA conversion function $\widehat{\mathcal{PDA}} : \mathsf{Exp} \to \mathbb{PDA}$. (Note the implicit use of the isomorphism $Q \times \widehat{\mathit{Kont}} \cong \widehat{\mathit{Conf}}$.)

At this point, we can answer questions about whether a specified control state is reachable by formulating a question about the intersection of a regular language with a context-free language described by the PDA. That is, if we want to know whether the control state $(e', \hat{\rho}, \hat{\sigma})$ is reachable in a program e, we can reduce the problem to determining:

$$\Sigma^* \cdot \left\{(e', \hat{\rho}, \hat{\sigma})\right\} \cdot \Sigma^* \cap \mathcal{L}(\widehat{\mathcal{PDA}}(e)) \neq \emptyset,$$

where $L_1 \cdot L_2$ is the concatenation of formal languages L_1 and L_2.

Theorem 5.1. *Control-state reachability is decidable.*

Proof. The intersection of a regular language and a context-free language is context-free. The emptiness of a context-free language is decidable. □

Now, consider how to use control-state reachability to answer the control-flow question from earlier. There are a finite number of possible control states in which the λ-term lam may flow to the function f in call site `(`f æ`)`; let's call the this set of states $\hat{S}$:

$$\hat{S} = \left\{ (\llbracket \texttt{(}f\ \text{æ}\texttt{)} \rrbracket, \hat{\rho}, \hat{\sigma}) : (lam, \hat{\rho}') \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \text{ for some } \hat{\rho}' \right\}.$$

What we want to know is whether any state in the set $\hat{S}$ is reachable in the PDA. In effect what we are asking is whether there exists a control state $q \in \hat{S}$ such that:

$$\Sigma^* \cdot \{q\} \cdot \Sigma^* \cap \mathcal{L}(\widehat{\mathcal{PDA}}(e)) \neq \emptyset.$$

If this is true, then lam may flow to f; if false, then it does not.

Problem: Doubly exponential complexity The non-emptiness-of-intersection approach establishes decidability of pushdown control-flow analysis. But, two exponential complexity barriers make this technique impractical.

First, there are an exponential number of both environments $(|\widehat{Addr}|^{|\mathsf{Var}|})$ and stores $(2^{|\widehat{Clo}| \times |\widehat{Addr}|})$ to consider for the set $\hat{S}$. On top of that, computing the intersection of a regular language with a context-free language will require enumeration of the (exponential) control-state-space of the PDA. As a result, this approach is doubly exponential. For the next few sections, our goal will be to lower the complexity of pushdown control-flow analysis.

6. Focusing on reachability

In the previous section, we saw that control-flow analysis reduces to the reachability of certain control states within a pushdown system. We also determined reachability by converting the abstracted CESK machine into a PDA, and using emptiness-testing on a language derived from that PDA. Unfortunately, we also found that this approach is deeply exponential.

Since control-flow analysis reduced to the reachability of control-states in the PDA, we skip the language problems and go directly to reachability algorithms of Bouajjani et al. [1997], Kodumal and Aiken [2004], Reps [1998] and Reps et al. [2005] that determine the reachable *configurations* within a pushdown system. These algorithms are even polynomial-time. Unfortunately, some of them are polynomial-time in the number of control states, and in the abstracted CESK machine, there are an exponential number of control states. We don't want to *enumerate* the entire control state-space, or else the search becomes exponential in even the best case.

To avoid this worst-case behavior, we present a straightforward pushdown-reachability algorithm that considers only the *reachable* control states. We cast our reachability algorithm as a fixed-point iteration, in which we incrementally construct the reachable subset of a pushdown system. We term these algorithms "iterative Dyck state graph construction."

A **Dyck state graph** is a compacted, rooted pushdown system $G = (S, \Gamma, E, s_0)$, 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:

$$(s_0, \langle\rangle) \xmapsto[G]{*} (s, \vec{\gamma}) \text{ for some stack } \vec{\gamma}.$$

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.[2]

We use $\mathbb{DSG}$ to denote the class of Dyck state graphs. Clearly:

$$\mathbb{DSG} \subset \mathbb{RPDS}.$$

A Dyck state graph is a rooted pushdown system with the "fat" trimmed off; in this case, unreachable control states and unreachable transitions are the "fat."

We can formalize the connection between rooted pushdown systems and Dyck state graphs with a map:

$$\mathcal{DSG} : \mathbb{RPDS} \to \mathbb{DSG}.$$

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

$$S = \left\{ q : (q_0, \langle\rangle) \xmapsto[M]{*} (q, \vec{\gamma}) \text{ for some stack } \vec{\gamma} \right\},$$

and the set E contains reachable edges:

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

and $s_0 = q_0$.

In practice, the real difference between a rooted pushdown system and a Dyck state graph is that our rooted pushdown system will be defined intensionally (having come from the components of an abstracted CESK machine), whereas the Dyck state graph will be defined extensionally, with the contents of each component explicitly enumerated during its construction.

Our near-term goals are (1) to convert our abstracted CESK machine into a rooted pushdown system and (2) to find an *efficient* method for computing an equivalent Dyck state graph from a rooted pushdown system.

To convert the abstracted CESK machine into a rooted pushdown system, we use the function $\widehat{\mathcal{RPDS}} : \mathsf{Exp} \to \mathbb{RPDS}$:

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

7. Compacting rooted pushdown systems

We now turn our attention to compacting a rooted pushdown system (defined intensionally) into a Dyck state graph (defined extension-

[2] 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.

ally). That is, we want to find 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 will provide a method (albeit an inefficient one) for computing the function $\mathcal{DSG}$. In the next section, we look at an optimized work-list driven algorithm that avoids the inefficiencies of this version.

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

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

Given a rooted pushdown system M, each application of the function $\mathcal{F}(M)$ accretes new edges at the frontier of the Dyck state graph. Once the algorithm reaches a fixed point, the Dyck state graph is complete:

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

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

To show $\mathcal{DSG}(M) \subseteq \mathrm{lfp}(\mathcal{F}(M))$, suppose this is not the case. Then, there must be at least one edge in $\mathcal{DSG}(M)$ that is not in $\mathrm{lfp}(\mathcal{F}(M))$. Let (s, g, s') be one such edge, such that the state s *is* in $\mathrm{lfp}(\mathcal{F}(M))$. 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 $\mathrm{lfp}(\mathcal{F}(M))$, which is a contradiction. Hence, $\mathcal{DSG}(M) \subseteq \mathrm{lfp}(\mathcal{F}(M))$. □

7.1 Complexity: Polynomial and exponential

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. Since there are at most $2|\Gamma|m^2 + m^2$ edges in the final graph, the maximum number of iterations is $2|\Gamma|m^2 + m^2$.

The cost of computing each iteration is harder to bound. The cost of determining whether to add a push edge is proportional to the size of the stack alphabet, while the cost of determining whether to add an ϵ-edge is constant, so the cost of determining all new push and pop edges to add is proportional to $|\Gamma|m + m$. 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 [Melski and Reps 2000] at each node, the cost of which is $O(|\Gamma_\pm|^3 m^3)$ [Kodumal and Aiken 2004].

To summarize, in terms of the number of reachable control states, the complexity of the most recent algorithm is:

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

While this approach is polynomial in the number of reachable control states, it is far from efficient. In the next section, we provide an optimized version of this fixed-point algorithm that maintains a work-list and an ϵ-closure graph to avoid spurious recomputation.

8. Efficiency: Work-lists and ϵ-closure graphs

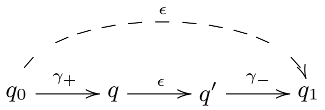
We have developed a fixed-point formulation of the Dyck state graph construction algorithm, but found that, in each iteration, it wasted effort by passing over all discovered states and edges, even though most will not contribute new states or edges. Taking a cue from graph search, we can adapt the fixed-point algorithm with a work-list. That is, our next algorithm will keep a work-list of new states and edges to consider, instead of reconsidering all of them. In each iteration, it will pull new states and edges from the work list, insert them into the Dyck state graph and then populate the work-list with new states and edges that have to be added as a consequence of the recent additions.

8.1 ϵ-closure graphs

Figuring out what edges to add as a consequence of another edge requires care, for adding an edge can have ramifications on distant control states. Consider, for example, adding the ϵ-edge $q \rightarrowtail^{\epsilon} q'$ into the following graph:

$$q_0 \xrightarrow{\gamma_+} q \qquad\qquad q' \xrightarrow{\gamma_-} q_1$$

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



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

To keep track of these implicit edges, we will construct a second graph in conjunction with the Dyck state graph: an ϵ-closure graph. In the ϵ-closure graph, every edge indicates the existence of a no-net-stack-change path between control states. The ϵ-closure graph simplifies the task of figuring out which states and edges are impacted by the addition of a new edge.

Formally, an **ϵ-closure graph**, is a pair $G_\epsilon = (N, H)$, where N is a set of states, and $H \subseteq N \times N$ is a set of edges. Of course, all ϵ-closure graphs are reflexive: every node has a self loop. We use the symbol $\mathbb{ECG}$ to denote the class of all ϵ-closure graphs.

We have two notations for finding ancestors and descendants of a state in an ϵ-closure graph $G_\epsilon = (N, H)$:

$$\begin{aligned}
\overleftarrow{G}_\epsilon[s] &= \left\{ s' : (s', s) \in H \right\} && \text{[ancestors]}\\
\overrightarrow{G}_\epsilon[s] &= \left\{ s' : (s, s') \in H \right\} && \text{[descendants].}
\end{aligned}$$

8.2 Integrating a work-list

Since we only want to consider new states and edges in each iteration, we need a work-list, or in this case, two work-graphs. A Dyck state work-graph is a pair $(\Delta S, \Delta E)$ in which the set ΔS contains a set of states to add, and the set ΔE contains edges to be added to a Dyck state graph.[3] We use $\Delta\mathbb{DSG}$ to refer to the class of all Dyck state work-graphs.

An ϵ-closure work-graph is a set ΔH of new ϵ-edges. We use $\Delta\mathbb{ECG}$ to refer to the class of all ϵ-closure work-graphs.

8.3 A new fixed-point iteration-space

Instead of consuming a Dyck state graph and producing a Dyck state graph, the new fixed-point iteration function will consume and produce a Dyck state graph, an ϵ-closure graph, a Dyck state work-graph and an ϵ-closure work graph. Hence, the iteration space of

[3] Technically, a work-graph is not an actual graph, since $\Delta E \not\subseteq \Delta S \times \Gamma_\pm \times \Delta S$; a work-graph is just a set of nodes and a set of edges.

$$\begin{aligned}
\mathcal{F}'(M) &= f, \text{ where}\\
M &= (Q, \Gamma, \delta, q_0)\\
f(G, G_\epsilon, \Delta G, \Delta H) &= (G', G'_\epsilon, \Delta G', \Delta H' - H), \text{ where}\\
(S, \Gamma, E, s_0) &= G\\
(S, H) &= G_\epsilon\\
(\Delta S, \Delta E) &= \Delta G\\
(\Delta E_0, \Delta H_0) &= \bigcup_{s \in \Delta S} \mathit{sprout}_M(s)\\
(\Delta E_1, \Delta H_1) &= \bigcup_{(s,\gamma_+,s') \in \Delta E} \mathit{addPush}_M(G, G_\epsilon)(s, \gamma_+, s')\\
(\Delta E_2, \Delta H_2) &= \bigcup_{(s,\gamma_-,s') \in \Delta E} \mathit{addPop}_M(G, G_\epsilon)(s, \gamma_-, s')\\
(\Delta E_3, \Delta H_3) &= \bigcup_{(s,\epsilon,s') \in \Delta E} \mathit{addEmpty}_M(G, G_\epsilon)(s, s')\\
(\Delta E_4, \Delta H_4) &= \bigcup_{(s,s') \in \Delta H} \mathit{addEmpty}_M(G, G_\epsilon)(s, s')\\
S' &= S \cup \Delta S\\
E' &= E \cup \Delta E\\
H' &= H \cup \Delta H\\
\Delta E' &= \Delta E_0 \cup \Delta E_1 \cup \Delta E_2 \cup \Delta E_3 \cup \Delta E_4\\
\Delta S' &= \left\{ s' : (s, g, s') \in \Delta E' \right\}\\
\Delta H' &= \Delta H_0 \cup \Delta H_1 \cup \Delta H_2 \cup \Delta H_3 \cup \Delta H_4\\
G' &= (S \cup \Delta S, \Gamma, E', q_0)\\
G'_\epsilon &= (S', H')\\
\Delta G' &= (\Delta S' - S', \Delta E' - E').
\end{aligned}$$

Figure 3. The fixed point of the function $\mathcal{F}'(M)$ contains the Dyck state graph of the rooted pushdown system M.

the new algorithm is:

$$\mathit{IDSG} = \mathbb{DSG} \times \mathbb{ECG} \times \Delta\mathbb{DSG} \times \Delta\mathbb{ECG}.$$

(The *I* in *IDSG* stands for *intermediate*.)

8.4 The ϵ-closure graph work-list algorithm

The function $\mathcal{F}' : \mathbb{RPDS} \to (\mathit{IDSG} \to \mathit{IDSG})$ generates the required iteration function (Figure 3). Please note that we implicitly distribute union across tuples:

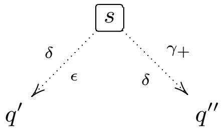
$$(X, Y) \cup (X', Y') = (X \cup X, Y \cup Y').$$

The functions *sprout*, *addPush*, *addPop*, *addEmpty* calculate the additional the Dyck state graph edges and ϵ-closure graph edges (potentially) introduced by a new state or edge.

Sprouting Whenever a new state gets added to the Dyck state graph, the algorithm must check whether that state has any new edges to contribute. Both push edges and ϵ-edges do not depend on the current stack, so any such edges for a state in the pushdown system's transition function belong in the Dyck state graph. The sprout function:

$$\mathit{sprout}_{(Q,\Gamma,\delta)} : Q \to (\mathcal{P}(\delta) \times \mathcal{P}(Q \times Q)),$$

checks whether a new state could produce any new push edges or no-change edges. We can represent its behavior diagrammatically:



which means if adding control state s:

add edge $s \rightarrowtail^{\epsilon} q'$ if it exists in δ, and

add edge $s \rightarrowtail^{\gamma_+} q''$ if it exists in δ.

Formally:

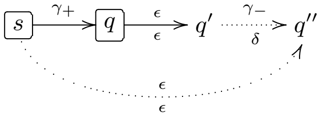
$$\mathit{sprout}_{(Q,\Gamma,\delta)}(s) = (\Delta E, \Delta H), \text{ where}$$

$$\Delta E = \left\{ s \overset{\epsilon}{\rightarrowtail} q : s \overset{\epsilon}{\rightarrowtail} q \in \delta \right\} \cup \left\{ s \overset{\gamma_+}{\rightarrowtail} q : s \overset{\gamma_+}{\rightarrowtail} q \in \delta \right\}$$
$$\Delta H = \left\{ s \rightarrowtail q : s \overset{\epsilon}{\rightarrowtail} q \in \delta \right\}.$$

Considering the consequences of a new push edge Once our algorithm adds a new push edge to a Dyck state graph, there is a chance that it will enable new pop edges for the same stack frame somewhere downstream. If and when it does enable pops, it will also add new edges to the ϵ-closure graph. The *addPush* function:

$$\mathit{addPush}_{(Q,\Gamma,\delta)} : \mathbb{DSG} \times \mathbb{ECG} \to \delta \to (\mathcal{P}(\delta) \times \mathcal{P}(Q \times Q)),$$

checks for ϵ-reachable states that could produce a pop. We can represent this action diagrammatically:



which means if adding push-edge $s \rightarrowtail^{\gamma_+} q$:

if pop-edge $q' \rightarrowtail^{\gamma_-} q''$ is in δ, then

add edge $q' \rightarrowtail^{\gamma_-} q''$, and

add ϵ-edge $s \rightarrowtail q''$.

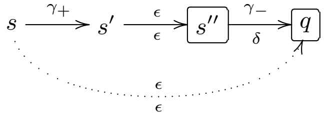
Formally:

$$\mathit{addPush}_{(Q,\Gamma,\delta)}(G, G_\epsilon)(s \overset{\gamma_+}{\rightarrowtail} q) = (\Delta E, \Delta H), \text{ where}$$
$$\Delta E = \left\{ q' \overset{\gamma_-}{\rightarrowtail} q'' : q' \in \overrightarrow{G}_\epsilon[q] \text{ and } q' \overset{\gamma_-}{\rightarrowtail} q'' \in \delta \right\}$$
$$\Delta H = \left\{ s \rightarrowtail q'' : q' \in \overrightarrow{G}_\epsilon[q] \text{ and } q' \overset{\gamma_-}{\rightarrowtail} q'' \in \delta \right\}.$$

Considering the consequences of a new pop edge Once the algorithm adds a new pop edge to a Dyck state graph, it will create at least one new ϵ-closure graph edge and possibly more by matching up with upstream pushes. The *addPop* function:

$$\mathit{addPop}_{(Q,\Gamma,\delta)} : \mathbb{DSG} \times \mathbb{ECG} \to \delta \to (\mathcal{P}(\delta) \times \mathcal{P}(Q \times Q)),$$

checks for ϵ-reachable push-edges that could match this pop-edge. We can represent this action diagrammatically:



which means if adding pop-edge $s'' \rightarrowtail^{\gamma_-} q$:

if push-edge $s \rightarrowtail^{\gamma_+} s'$ is already in the Dyck state graph, then add ϵ-edge $s \rightarrowtail q$.

Formally:

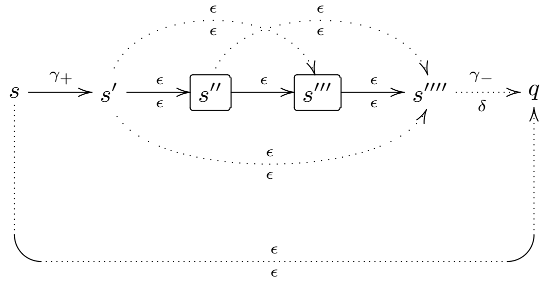
$$\mathit{addPop}_{(Q,\Gamma,\delta)}(G, G_\epsilon)(s'' \overset{\gamma_-}{\rightarrowtail} q) = (\Delta E, \Delta H), \text{ where}$$

$$\Delta E = \emptyset \text{ and } \Delta H = \left\{ s \rightarrowtail q : s' \in \overleftarrow{G}_\epsilon[s''] \text{ and } s \overset{\gamma_+}{\rightarrowtail} s' \in G \right\}.$$

Considering the consequences of a new ϵ-edge Once the algorithm adds a new ϵ-closure graph edge, it may transitively have to add more ϵ-closure graph edges, and it may connect an old push to (perhaps newly enabled) pop edges. The $\mathit{addEmpty}$ function:

$$\mathit{addEmpty}_{(Q,\Gamma,\delta)} : \\ \mathbb{DSG} \times \mathbb{ECG} \to (Q \times Q) \to (\mathcal{P}(\delta) \times \mathcal{P}(Q \times Q)),$$

checks for newly enabled pops and ϵ-closure graph edges: Once again, we can represent this action diagrammatically:



which means if adding ϵ-edge $s'' \rightarrowtail s'''$:

if pop-edge $s'''' \rightarrowtail^{\gamma_-} q$ is in δ, then

add ϵ-edge $s \rightarrowtail q$; and

add edge $s'''' \rightarrowtail^{\gamma_-} q$;

add ϵ-edges $s' \rightarrowtail s'''$, $s'' \rightarrowtail s''''$, and $s' \rightarrowtail s''''$.

Formally:

$$\mathit{addEmpty}_{(Q,\Gamma,\delta)}(G, G_\epsilon)(s'' \rightarrowtail s''') = (\Delta E, \Delta H), \text{ where}$$

$$\Delta E = \left\{ s'''' \overset{\gamma_-}{\rightarrowtail} q : s' \in \overleftarrow{G}_\epsilon[s''] \text{ and } s'''' \in \overrightarrow{G}_\epsilon[s'''] \text{ and } s \overset{\gamma_+}{\rightarrowtail} s' \in G \right\}$$

$$\begin{aligned}\Delta H = {} & \left\{ s \rightarrowtail q : s' \in \overleftarrow{G}_\epsilon[s''] \text{ and } s'''' \in \overrightarrow{G}_\epsilon[s'''] \text{ and } s \overset{\gamma_+}{\rightarrowtail} s' \in G \right\} \\ & \cup \left\{ s' \rightarrowtail s''' : s' \in \overleftarrow{G}_\epsilon[s''] \right\} \\ & \cup \left\{ s'' \rightarrowtail s'''' : s'''' \in \overrightarrow{G}_\epsilon[s'''] \right\} \\ & \cup \left\{ s' \rightarrowtail s'''' : s' \in \overleftarrow{G}_\epsilon[s''] \text{ and } s'''' \in \overrightarrow{G}_\epsilon[s'''] \right\}.\end{aligned}$$

8.5 Termination and correctness

Because the iteration function is no longer monotonic, we have to prove that a fixed point exists. It is trivial to show that the Dyck state graph component of the iteration-space ascends monotonically with each application; that is:

Lemma 8.1. *Given $M \in \mathbb{RPDS}, G \in \mathbb{DSG}$ such that $G \subseteq M$, if $\mathcal{F}'(M)(G, G_\epsilon, \Delta G) = (G', G'_\epsilon, \Delta G')$, then $G \subseteq G'$.*

Since the size of the Dyck state graph is bounded by the original pushdown system M, the Dyck state graph will eventually reach a fixed point. Once the Dyck state graph reaches a fixed point, both work-graphs/sets will be empty, and the ϵ-closure graph will also stabilize. We can also show that this algorithm is correct:

Theorem 8.1. $\mathrm{lfp}(\mathcal{F}'(M)) = (\mathcal{DSG}(M), G_\epsilon, (\emptyset, \emptyset), \emptyset)$.

Proof. The proof is similar in structure to the previous one. □

8.6 Complexity: Still exponential, but more efficient

As with the previous algorithm, 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 run-time of the algorithm is bounded by the size of the final Dyck state graph plus the size of the ϵ-closure graph. Suppose the final Dyck state graph has m states. In the worst case, the iteration function adds only a single edge each time. There are at most $2|\Gamma|m^2 + m^2$ edges in the Dyck state graph and at most m^2 edges in the ϵ-closure graph, which bounds the number of iterations.

Next, we must reason about the worst-case cost of adding an edge: how many edges might an individual iteration consider? In the worst case, the algorithm will consider every edge in every iteration, leading to an asymptotic time-complexity of:

$$O((2|\Gamma|m^2 + 2m^2)^2) = O(|\Gamma|^2 m^4).$$

While still high, this is a an improvement upon the previous algorithm. For sparse Dyck state graphs, this is a reasonable algorithm.

9. Polynomial-time complexity from widening

In the previous section, we developed a more efficient fixed-point algorithm for computing a Dyck state graph. Even with the core improvements we made, the algorithm remained exponential in the worst case, owing to the fact that there could be an exponential number of reachable control states. When an abstract interpretation is intolerably complex, the standard approach for reducing complexity and accelerating convergence is widening [Cousot and Cousot 1977]. (Of course, widening techniques trade away some precision to gain this speed.) It turns out that the small-step variants of finite-state CFAs are exponential without some sort of widening as well.

To achieve polynomial time complexity for pushdown control-flow analysis requires the same two steps as the classical case: (1) widening the abstract interpretation to use a global, "single-threaded" store and (2) selecting a monovariant allocation function to collapse the abstract configuration-space. Widening eliminates a source of exponentiality in the size of the store; monovariance eliminates a source of exponentiality from environments. In this section, we redevelop the pushdown control-flow analysis framework with a single-threaded store and calculate its complexity.

9.1 Step 1: Refactor the concrete semantics

First, consider defining the reachable states of the concrete semantics using fixed points. That is, let the system-space of the evaluation function be sets of configurations:

$$C \in \mathit{System} = \mathcal{P}(\mathit{Conf}) = \mathcal{P}(\mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont}).$$

We can redefine the concrete evaluation function:

$$\mathcal{E}(e) = \mathrm{lfp}(f_e), \text{ where } f_e : \mathit{System} \to \mathit{System} \text{ and}$$
$$f_e(C) = \{\mathcal{I}(e)\} \cup \left\{ c' : c \in C \text{ and } c \Rightarrow c' \right\}.$$

9.2 Step 2: Refactor the abstract semantics

We can take the same approach with the abstract evaluation function, first redefining the abstract system-space:

$$\begin{aligned}\hat{C} \in \widehat{\mathit{System}} &= \mathcal{P}\left(\widehat{\mathit{Conf}}\right) \\ &= \mathcal{P}\left(\mathsf{Exp} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Kont}}\right),\end{aligned}$$

and then the abstract evaluation function:

$$\hat{\mathcal{E}}(e) = \text{lfp}(\hat{f}_e), \text{ where } \hat{f}_e : \widehat{System} \to \widehat{System} \text{ and}$$
$$\hat{f}_e(\hat{C}) = \left\{\hat{\mathcal{I}}(e)\right\} \cup \left\{\hat{c}' : \hat{c} \in \hat{C} \text{ and } \hat{c} \leadsto \hat{c}'\right\}.$$

What we'd like to do is shrink the abstract system-space with a refactoring that corresponds to a widening.

9.3 Step 3: Single-thread the abstract store

We can approximate a set of abstract stores $\{\hat{\sigma}_1, \ldots, \hat{\sigma}_n\}$ with the least-upper-bound of those stores: $\hat{\sigma}_1 \sqcup \cdots \sqcup \hat{\sigma}_n$. We can exploit this by creating a new abstract system space in which the store is factored out of every configuration. Thus, the system-space contains a set of *partial configurations* and a single global store:

$$\widehat{System}' = \mathcal{P}\left(\widehat{PConf}\right) \times \widehat{Store}$$
$$\hat{\pi} \in \widehat{PConf} = \mathsf{Exp} \times \widehat{Env} \times \widehat{Kont}.$$

We can factor the store out of the abstract transition relation as well, so that $(\overset{\hat{\sigma}}{\rightarrowtriangle}) \subseteq \widehat{PConf} \times (\widehat{PConf} \times \widehat{Store})$:

$$(e, \hat{\rho}, \hat{\kappa}) \overset{\hat{\sigma}}{\rightarrowtriangle} ((e', \hat{\rho}', \hat{\kappa}'), \hat{\sigma}') \text{ iff } (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \leadsto (e', \hat{\rho}', \hat{\sigma}', \hat{\kappa}'),$$

which gives us a new iteration function, $\hat{f}'_e : \widehat{System}' \to \widehat{System}'$,

$$\hat{f}'_e(\hat{P}, \hat{\sigma}) = (\hat{P}', \hat{\sigma}'), \text{ where}$$
$$\hat{P}' = \left\{\hat{\pi}' : \hat{\pi} \overset{\hat{\sigma}}{\rightarrowtriangle} (\hat{\pi}', \hat{\sigma}'')\right\} \cup \{\hat{\pi}_0\}$$
$$\hat{\sigma}' = \bigsqcup \left\{\hat{\sigma}'' : \hat{\pi} \overset{\hat{\sigma}}{\rightarrowtriangle} (\hat{\pi}', \hat{\sigma}'')\right\}$$
$$(\hat{\pi}_0, \langle\rangle) = \hat{\mathcal{I}}(e).$$

9.4 Step 4: Dyck state control-flow graphs

Following the earlier Dyck state graph reformulation of the pushdown system, we can reformulate the set of partial configurations as a *Dyck state control-flow graph*. A **Dyck state control-flow graph** is a frame-action-labeled graph over partial control states, and a **partial control state** is an expression paired with an environment:

$$\widehat{System}'' = \widehat{DSCFG} \times \widehat{Store}$$
$$\widehat{DSCFG} = \mathcal{P}(\widehat{PState}) \times \mathcal{P}(\widehat{PState} \times \widehat{Frame}_{\pm} \times \widehat{PState})$$
$$\hat{\psi} \in \widehat{PState} = \mathsf{Exp} \times \widehat{Env}.$$

In a Dyck state control-flow graph, the partial control states are partial configurations which have dropped the continuation component; the continuations are encoded as paths through the graph.

If we wanted to do so, we could define a new monotonic iteration function analogous to the simple fixed-point formulation of Section 7:

$$\hat{f}_e : \widehat{System}'' \to \widehat{System}'',$$

again using CFL-reachability to add pop edges at each step.

A preliminary analysis of complexity Even without defining the system-space iteration function, we can ask, *How many iterations will it take to reach a fixed point in the worst case?* This question is really asking, *How many edges can we add?* And, *How many entries are there in the store?* Summing these together, we arrive at the worst-case number of iterations:

$$\overbrace{|\widehat{PState}| \times |\widehat{Frame}_{\pm}| \times |\widehat{PState}|}^{\text{DSCFG edges}} + \overbrace{|\widehat{Addr}| \times |\widehat{Clo}|}^{\text{store entries}}.$$

With a monovariant allocation scheme that eliminates abstract environments, the number of iterations ultimately reduces to:

$$|\mathsf{Exp}| \times (2|\widehat{\mathsf{Var}}| + 1) \times |\mathsf{Exp}| + |\mathsf{Var}| \times |\mathsf{Lam}|,$$

which means that, in the worst case, the algorithm makes a cubic number of iterations with respect to the size of the input program.[4]

The worst-case cost of the each iteration would be dominated by a CFL-reachability calculation, which, in the worst case, must consider every state and every edge:

$$O(|\mathsf{Var}|^3 \times |\mathsf{Exp}|^3).$$

Thus, each iteration takes $O(n^6)$ and there are a maximum of $O(n^3)$ iterations, where n is the size of the program. So,total complexity would be $O(n^9)$ for a monovariant pushdown control-flow analysis with this scheme, where n is again the size of the program. Although this algorithm is polynomial-time, we can do better.

9.5 Step 5: Reintroduce ϵ-closure graphs

Replicating the evolution from Section 8 for this store-widened analysis, we arrive at a more efficient polynomial-time analysis. An ϵ-closure graph in this setting is a set of pairs of store-less, continuation-less partial states:

$$\widehat{ECG} = \mathcal{P}\left(\widehat{PState} \times \widehat{PState}\right).$$

Then, we can set the system space to include ϵ-closure graphs:

$$\widehat{System}''' = \widehat{DSG} \times \widehat{ECG} \times \widehat{Store}.$$

Before we redefine the iteration function, we need another factored transition relation. The stack- and action-factored transition relation $(\rightharpoondown^{\hat{\sigma}}_{g}) \subseteq \widehat{PState} \times \widehat{PState} \times Store$ determines if a transition is possible under the specified store and stack-action:

$$(e, \hat{\rho}) \underset{\hat{\phi}_+}{\overset{\hat{\sigma}}{\rightharpoondown}} ((e', \hat{\rho}'), \hat{\sigma}') \text{ iff } (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \leadsto (e', \hat{\rho}', \hat{\sigma}', \hat{\phi} : \hat{\kappa}')$$
$$(e, \hat{\rho}) \underset{\hat{\phi}_-}{\overset{\hat{\sigma}}{\rightharpoondown}} ((e', \hat{\rho}'), \hat{\sigma}') \text{ iff } (e, \hat{\rho}, \hat{\sigma}, \hat{\phi} : \hat{\kappa}) \leadsto (e', \hat{\rho}', \hat{\sigma}', \hat{\kappa}')$$
$$(e, \hat{\rho}) \underset{\epsilon}{\overset{\hat{\sigma}}{\rightharpoondown}} ((e', \hat{\rho}'), \hat{\sigma}') \text{ iff } (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \leadsto (e', \hat{\rho}', \hat{\sigma}', \hat{\kappa}').$$

Now, we can redefine the iteration function (Figure 4).

Theorem 9.1. *Pushdown 0CFA can be computed in $O(n^6)$-time, where n is the size of the program.*

Proof. As before, the maximum number of iterations is cubic in the size of the program for a monovariant analysis. Fortunately, the cost of each iteration is also now bounded by the number of edges in the graph, which is also cubic. □

10. Applications

Pushdown control-flow analysis offers more precise control-flow analysis results than the classical finite-state CFAs. Consequently, pushdown control-flow analysis improves flow-driven optimizations (*e.g.*, constant propagation, global register allocation, inlining [Shivers 1991]) by eliminating more of the false positives that block their application.

The more compelling applications of pushdown control-flow analysis are those which are difficult to drive with classical control-flow analysis. Perhaps not surprisingly, the best examples of such

[4] In computing the number of frames, we note that in every continuation, the variable and the expression uniquely determine each other based on the let-expression from which they both came. As a result, the number of abstract frames available in a monovariant analysis is bounded by both the number of variables and the number of expressions, *i.e.*, $|\widehat{Frame}| = |\mathsf{Var}|$.

$$\hat{f}((\hat{P}, \hat{E}), \hat{H}, \hat{\sigma}) = ((\hat{P}', \hat{E}'), \hat{H}', \hat{\sigma}''), \text{ where}$$

$$\begin{aligned}
\hat{T}_+ &= \left\{ (\hat{\psi} \overset{\hat{\phi}_+}{\rightarrowtail} \hat{\psi}', \hat{\sigma}') : \hat{\psi} \underset{\hat{\phi}_+}{\overset{\hat{\sigma}}{\rightharpoondown}} (\hat{\psi}', \hat{\sigma}') \right\} \\
\hat{T}_\epsilon &= \left\{ (\hat{\psi} \overset{\epsilon}{\rightarrowtail} \hat{\psi}', \hat{\sigma}') : \hat{\psi} \underset{\epsilon}{\overset{\hat{\sigma}}{\rightharpoondown}} (\hat{\psi}', \hat{\sigma}') \right\} \\
\hat{T}_- &= \{ (\hat{\psi}'' \overset{\hat{\phi}_-}{\rightarrowtail} \hat{\psi}''', \hat{\sigma}') : \hat{\psi}'' \underset{\hat{\phi}_-}{\overset{\hat{\sigma}}{\rightharpoondown}} (\hat{\psi}''', \hat{\sigma}') \text{ and} \\
&\qquad\qquad\qquad\qquad \hat{\psi} \overset{\hat{\phi}_+}{\rightarrowtail} \hat{\psi}' \in \hat{E} \text{ and} \\
&\qquad\qquad\qquad\qquad \hat{\psi}' \rightarrowtail \hat{\psi}'' \in \hat{H} \} \\
\hat{T}' &= \hat{T}_+ \cup \hat{T}_\epsilon \cup \hat{T}_- \\
\hat{E}' &= \left\{ \hat{e} : (\hat{e}, \_) \in \hat{T}' \right\} \\
\hat{\sigma}'' &= \bigsqcup \left\{ \hat{\sigma}' : (\_, \hat{\sigma}') \in \hat{T}' \right\} \\
\hat{H}_\epsilon &= \left\{ \hat{\psi} \rightarrowtail \hat{\psi}'' : \hat{\psi} \rightarrowtail \hat{\psi}' \in \hat{H} \text{ and } \hat{\psi}' \rightarrowtail \hat{\psi}'' \in \hat{H} \right\} \\
\hat{H}_{+-} &= \{ \hat{\psi} \rightarrowtail \hat{\psi}''' : \hat{\psi} \overset{\hat{\phi}_+}{\rightarrowtail} \hat{\psi}' \in \hat{E} \text{ and } \hat{\psi}' \rightarrowtail \hat{\psi}'' \in \hat{H} \\
&\qquad\qquad \text{and } \hat{\psi}'' \overset{\hat{\phi}_-}{\rightarrowtail} \hat{\psi}''' \in \hat{E} \} \\
\hat{H}' &= \hat{H}_\epsilon \cup \hat{H}_{+-} \\
\hat{P}' &= \hat{P} \cup \left\{ \hat{\psi}' : \hat{\psi} \overset{g}{\rightarrowtail} \hat{\psi}' \right\}.
\end{aligned}$$

Figure 4. An ϵ-closure graph-powered iteration function for pushdown control-flow analysis with a single-threaded store.

analyses are escape analysis and interprocedural dependence analysis. Both of these analyses are limited by a static analyzer's ability to reason about the stack, the core competency of pushdown control-flow analysis. (We leave an in-depth formulation and study of these analyses to future work.)

10.1 Escape analysis

In escape analysis, the objective is to determine whether a heap-allocated object is safely convertible into a stack-allocated object. In other words, the compiler is trying to figure out whether the frame in which an object is allocated outlasts the object itself. In higher-order languages, closures are candidates for escape analysis.

Determining whether all closures over a particular λ-term lam may be heap-allocated is straightforward: find the control states in the Dyck state graph in which closures over lam are being created, then find all control states reachable from these states over only ϵ-edge and push-edge transitions. Call this set of control states the "safe" set. Now find all control states which are invoking a closure over lam. If any of these control states lies outside of the safe set, then stack-allocation may not be safe; if, however, all invocations lie within the safe set, then stack-allocation of the closure is safe.

10.2 Interprocedural dependence analysis

In interprocedural dependence analysis, the goal is to determine, for each λ-term, the set of resources which it may read or write when it is called. Might and Prabhu showed that if one has knowledge of the program stack, then one can uncover interprocedural dependencies [Might and Prabhu 2009]. We can adapt that technique to work with Dyck state graphs. For each control state, find the set of reachable control states along only ϵ-edges and pop-edges. The frames on the pop-edges determine the frames which could have been on the stack when in the control state. The frames that are live on the stack determine the procedures that are live on the stack. Every procedure that is live on the stack has a read-dependence on any resource being read in the control state, while every procedure that is live on the stack also has a write-dependence on any resource being written in the control state. This logic is the direct complement of "if f calls g and g accesses a, then f also accesses a."

11. Related work

Pushdown control-flow analysis draws on work in higher-order control-flow analysis [Shivers 1991], abstract machines [Felleisen and Friedman 1987] and abstract interpretation [Cousot and Cousot 1977].

Context-free analysis of higher-order programs The closest related work for this is Vardoulakis and Shivers very recent work on CFA2 [Vardoulakis and Shivers 2010]. 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 alludes to exploiting context-free languages, context-free languages are not explicit in its formulation in the same way that pushdown systems are in pushdown control-flow analysis. With respect to CFA2, pushdown control-flow analysis is polyvariant, covers direct-style, and the monovariant instatiation is lower in complexity (CFA2 is exponential-time).

On the other hand, CFA2 distinguishes stack-allocated and store-allocated variable bindings, whereas our formulation of pushdown control-flow analysis does not and 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.

Calculation approach to abstract interpretation Midtgaard and Jensen [2009] systematically calculate 0CFA using the Cousot-Cousot-style calculational approach to abstract interpretation [Cousot 1999] applied to an ANF λ-calculus. Like the present work, Midtgaard and Jensen start with the CESK machine of Flanagan et al. [1993] 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 analsysis 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 [Bouajjani et al. 1997, Kodumal and Aiken 2004, Reps 1998, Reps et al. 2005]. For instance, ϵ-closure graphs, or equivalent variants thereof, appear in many context-free-language and pushdown reachability algorithms. For the less efficient versions of our analyses, 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 [Melski and Reps 2000] and type-based polymorphic control-flow analysis [Rehof and Fähndrich 2001]. 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. In addition, our technique is not restricted to typed programs.

Model-checking higher-order recursion schemes There is terminology overlap with work by Kobayashi [2009] 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 [2006] 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. Moreover, there does not seem to be a polynomial-time variant of Kobyashi's method.

Other escape and dependence analyses We presented escape and dependence analyses to prove a point: that pushdown control-flow analysis is more powerful than classical control-flow analysis, in the sense that it can answer different kinds of questions. We have not yet compared our analyses with the myriad escape and dependence analyses (*e.g.*, [Blanchet 1998]) that exist in the literature, though we do expect that, with their increased precision, our analyses will be strongly competitive.

12. Conclusion

Pushdown control-flow analysis is an alternative paradigm for the analysis of higher-order programs. By modeling the run-time program stack with the stack of a pushdown system, pushdown control-flow analysis precisely matches returns to their calls. We derived pushdown control-flow analysis as an abstract interpretation of a CESK machine in which its stack component is left unbounded. As this abstract interpretation ranged over an infinite state-space, we sought a decidable method for determining th reachable states. We found one by converting the abstracted CESK into a PDA that recognized the language of legal control-state sequences. By intersecting this language with a specific regular language and checking non-emptiness, we were able to answer control-flow questions. From the PDA formulation, we refined the technique to reduce complexity from doubly exponential, to best-case exponential, to worst-case exponential, to polynomial. We ended with an efficient, polyvariant and precise framework.

Future work Pushdown control-flow analysis exploits the fact that clients of static analyzers often need information about control states rather than stacks. Should clients require information about complete configurations—control states plus stacks—our analysis is lacking. Our framework represents configurations as *paths* through Dyck state graphs. Its results can provide a regular description of the stack, but at a cost proportional to the size of the graph. For a client like abstract garbage collection, which would pay this cost for every edge added to the graph, this cost is unacceptable. Our future work will examine how to incrementally summarize stacks paired with each control state during the analysis.

Acknowledgements We thank Dimitrios Vardoulakis for comments and discussions, and reviewers of ICFP and this workshop.

References

Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–37, New York, NY, USA, 1998. ACM.

Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory*, pages 135–150, London, UK, 1997. Springer-Verlag.

Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

Patrick Cousot and Radhia Cousot. 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*, pages 238–252, New York, NY, USA, 1977. ACM Press.

Matthias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 314+, New York, NY, USA, 1987. ACM.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming Language Design and Implementation*, volume 28, pages 237–247, New York, NY, USA, June 1993. ACM.

Naoki Kobayashi. 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*, pages 416–428, New York, NY, USA, 2009. ACM.

John Kodumal and Alex Aiken. The set constraint/CFL reachability connection in practice. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 207–218, New York, NY, USA, 2004. ACM.

David Melski and Thomas W. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1-2):29–98, October 2000.

Jan Midtgaard and Thomas P. Jensen. 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*, pages 287–298. ACM, 2009.

Matthew Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, June 2007.

Matthew Might and Tarun Prabhu. Interprocedural dependence analysis of higher-order programs via stack reachability. In *Proceedings of the 2009 Workshop on Scheme and Functional Programming*, Boston, Massachussetts, USA, 2009.

C. H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 81–90. IEEE, 2006.

Jakob Rehof and Manuel Fähndrich. 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*, pages 54–66, New York, NY, USA, 2001. ACM.

Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, December 1998.

Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1-2):206–263, 2005.

Olin G. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 2 edition, February 2005.

David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2010.

Dimitrios Vardoulakis and Olin Shivers. CFA2: a Context-Free Approach to Control-Flow Analysis. In *European Symposium on Programming (ESOP)*, volume 6012 of *LNCS*, pages 570–589. Springer, 2010.