Under consideration for publication in J. Functional Programming

Introspective pushdown analysis

CHRISTOPHER EARL
University of Utah

ILYA SERGEY
IMDEA Software Institute

J. IAN JOHNSON
Northeastern University

MATTHEW MIGHT
University of Utah

DAVID VAN HORN
Northeastern University

Abstract

In the static analysis of functional programs, pushdown flow analysis and abstract garbage collection skirt just inside the boundaries of soundness and decidability. This work illuminates and conquers the theoretical challenges that stand in the way of combining the power of these techniques. Pushdown flow analysis grants unbounded yet computable polyvariance to the analysis of return-flow in higher-order programs. Abstract garbage collection grants unbounded polyvariance to abstract addresses which become unreachable between invocations of the abstract contexts in which they were created. Pushdown analysis solves the problem of precisely analyzing recursion in higher-order languages; abstract garbage collection is essential in solving the "stickiness" problem. Alone, each method reduces analysis times and boosts precision by orders of magnitude.

We combine these methods. 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. Pushdown flow analysis already skirts just inside the boundary of decidability, so it is not clear that integrating abstract garbage collection will preserve decidability. In fact, introducing abstract garbage collection breaks the pushdown framework: to find the root set of reachable addresses, the analysis must consider the full contents of all possible stacks associated with a given control state. Such introspection clearly violates the constraints on a pushdown system.

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.

```
(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)))
```

Fig. 1. A small example to illuminate the strengths and weaknesses of both pushdown analysis and abstract garbage collection.

1 Introduction

The development of a context-free[1] approach to control-flow analysis (CFA2) by Vardoulakis and Shivers provoked a shift in the static analysis of higher-order programs (Vardoulakis and Shivers 2010). Prior to CFA2, a precise analysis of recursive behavior had been a challenge—even though flow analyses have an important role to play in optimization for functional languages, such as flow-driven inlining (Might and Shivers 2006a), interprocedural constant propagation (Shivers 1991) and type-check elimination (Wright and Jagannathan 1998).

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. Its payoff is significant reductions in analysis time *as a result of* corresponding increases in precision.

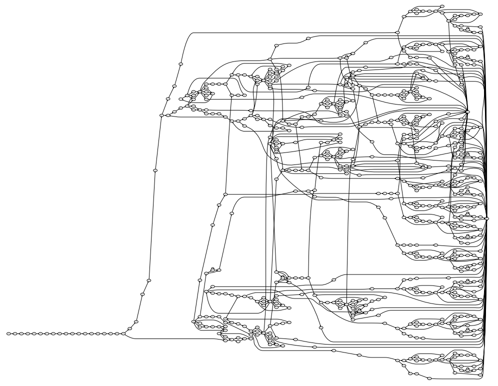
For a visual measure of the impact, Figure 2 renders the abstract transition graph (a model of all possible traces through the program) for the toy program in Figure 1. For this example, pushdown analysis eliminates spurious return-flow from the use of recursion. But, recursion is just one problem of 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 (Might and Shivers 2006b) were developed to solve the cross-flow problem.[2] In fact, abstract garbage collection, by itself, also delivers significant improvements to analytic speed and precision. (See Figure 2 again for a visualization of that impact.)

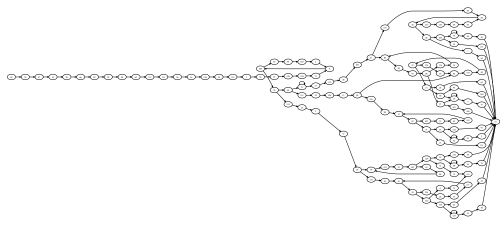
It is natural to ask: can abstract garbage collection and pushdown analysis work together? Can their strengths be multiplied? At first, the answer appears to be a disheartening "*No*."

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

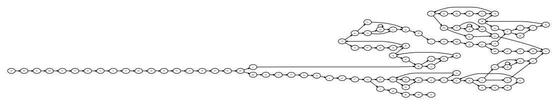
[2] The cross-flow problem arises because monotonicity prevents revoking a judgment like "procedure `f` flows to `x`," or "procedure `g` flows to `x`," once it's been made.



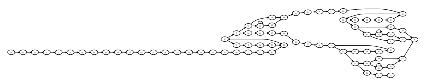
(1) without pushdown analysis or abstract GC: 653 states



(2) with pushdown only: 139 states



(3) with GC only: 105 states



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

Fig. 2. We generated an abstract transition graph for the same program from Figure 1 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.

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 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—that provide read-only access to the stack. Control-state reachability for the straightforward formulation of these systems ends up being uncomputable (but barely). By introducing a relatively weak monotonicity constraint on transitions, introspective pushdown systems have 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 2 (and echoed by later benchmarks) conveys the impact of this fusion.

Secondary motivations There are four secondary motivations for this work: (1) bringing context-sensitivity to pushdown analysis; (2) exposing the context-freedom of the analysis; (3) enabling pushdown analysis without continuation-passing style; and (4) unambiguously defining an algorithm for computing pushdown analysis, introspectively or otherwise.

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. Our development exposes the classical parameters (exposed as allocation functions in a semantics) that allow one to tune the context-sensitivity and polyvariance.

In addition, the context-freedom of CFA2 is buried implicitly inside an imperative summarization algorithm. No pushdown system or context-free grammar is explicitly identified. Thus, a necessary precursor to our work was to make the pushdown system in CFA2 explicit, and to make the control-state reachability algorithm purely functional.

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. By abstracting all machine components except for the program stack, it converts naturally and readily into a pushdown system.

Finally, to bring much-needed clarity to algorithmic formualtion of pushdown analysis, we have inlined a reference implementation in Haskell. We have kept the code as close in form to the mathematics as possible, so that where concessions are made to the implementation, they are obvious.

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. Uncharacteristically, we leave the stack component—the K(ontinuation)—unbounded.

A shift in perspective reveals that this abstract interpretation is a pushdown system. We encode it as a pushdown automaton explicitly, and pose control state reachability as a decidable language intersection problem. We then extract a rooted pushdown system from the pushdown automaton. For completeness, we fully develop pushdown analysis for higher-order programs, including an efficient algorithm for computing reachable control states. We go further by characterizing complexity and demonstrating the approximations necessary to get to a polynomial-time algorithm.

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 when introspective pushdown systems are subjected to a straightforward monotonicity constraint.

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.

[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.

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.
6. We provide a mirror of the major formal development as working Haskell code. This code illuminates dark corners of pushdown analysis and it provides a concise formal reference implementation.

2 Pushdown preliminaries

The literature contains many equivalent definitions of pushdown machines, so we adapt our own definitions from Sipser (Sipser 2005). *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:

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

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:

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

In this paper, the symbol g represents some stack action. In Haskell, we can turn any data type in a stack-action alphabet:

```
data StackAct frame = Push { frame :: frame }
                    | Pop  { frame :: frame }
                    | Unch
```

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:

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

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:

$$(q,\vec{\gamma}) \underset{M}{\overset{\varepsilon}{\longmapsto}} (q',\vec{\gamma}) \text{ iff } q \overset{\varepsilon}{\rightarrowtail} q' \in \delta \qquad \text{[no change]}$$

$$(q,\gamma:\vec{\gamma}) \underset{M}{\overset{\gamma_-}{\longmapsto}} (q',\vec{\gamma}) \text{ iff } q \overset{\gamma_-}{\rightarrowtail} q' \in \delta \qquad \text{[pop]}$$

$$(q,\vec{\gamma}) \underset{M}{\overset{\gamma_+}{\longmapsto}} (q',\gamma:\vec{\gamma}) \text{ iff } q \overset{\gamma_+}{\rightarrowtail} q' \in \delta \qquad \text{[push].}$$

- 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:

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

In Haskell, we will need to two functional encodings of δ:

```
type Delta control frame =
 (TopDelta control frame, NopDelta control frame)
type TopDelta control frame =
 control -> frame -> [(control,StackAct frame)]
type NopDelta control frame =
 control -> [(control,StackAct frame)]
```

If we only want to know push and no-change transitions, we can find these with a `NopDelta` function without providing the frame that is currently on top of the stack. If we want pop transitions as well, we can find these with a `TopDelta` function, but of course, it must have access to the top of the stack. In practice, a `TopDelta` function would suffice, but there are situations where only push and no-change transitions are needed, and having access to `NopDelta` avoids extra computation.

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

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

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:

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

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:

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

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 (Bouajjani et al. 1997; Kodumal and Aiken 2004a; Reps 1998; Reps et al. 2005). We will develop a complete alternative to summarization, and then instrument this development for introspective pushdown systems.

2.6 Pushdown automata

A **pushdown automaton** is an input-accepting 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. $\delta \subseteq Q \times \Gamma_{\pm} \times (\Sigma \cup \{\varepsilon\}) \times Q$ 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 $\mathscr{L}(M) \subseteq \Sigma^*$:

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

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

2.7 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;
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ 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 $\mathbb{NFA}$.

In Haskell, we represent an NFA as a set of labeled forward edges, the inverse of those edges (for convenience), a start state and an end state:

```
type NFA state char =
 (NFAEdges state char,NFAEdges state char,state,state)
type NFAEdges state char = state :-> ℙ(Maybe char,state)
```

2.8 Transliterating formalism into Haskell

Where it is critical to understanding the details of the analysis, we have transliterated the formalism into Haskell. We make use of a two extensions in GHC:

```
-XTypeOperators -XTypeSynonymInstances
```

All code is in the context of the following header:

```
import Prelude hiding ((!!))

import Data.Map as Map hiding (map,foldr)
import Data.Set as Set hiding (map,foldr)
import Data.List as List hiding ((!!))

type ℙ s = Set.Set s
type k :-> v = Map k v

(==>) :: a -> b -> (a,b)
(==>) x y = (x,y)

(//) :: Ord a => (a :-> b) -> [(a,b)] -> (a :-> b)
(//) f [(x,y)] = Map.insert x y f

set x = Set.singleton x
```

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:

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

We can transliterate this structure in Haskell:

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

Fig. 3. The concrete configuration-space.

```
data Exp    = Ret AExp
            | App Call
            | Let1 Var Call Exp
data AExp   = Ref Var
            | Lam Lambda
data Lambda = Var :=> Exp
data Call   = AExp :@ AExp
type Var    = String
```

And, we'll assume the standard instances of type classes like `Ord` and `Eq`.

We use the CESK machine of Felleisen and Friedman (Felleisen and Friedman 1987) 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).

3.1 Semantics

To define the semantics, we need five items:

1. $\mathscr{I} : \mathsf{Exp} \to \mathit{Conf}$ injects an expression into a configuration:
$$c_0 = \mathscr{I}(e) = (e, [], [], \langle\rangle).$$
2. $\mathscr{A} : \mathsf{Atom} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Clo}$ evaluates atomic expressions:
$$
\begin{aligned}
\mathscr{A}(\mathit{lam}, \rho, \sigma) &= (\mathit{lam}, \rho) && \text{[closure creation]}\\
\mathscr{A}(v, \rho, \sigma) &= \sigma(\rho(v)) && \text{[variable look-up].}
\end{aligned}
$$
3. $(\Rightarrow) \subseteq \mathit{Conf} \times \mathit{Conf}$ transitions between configurations. (Defined below.)
4. $\mathscr{E} : \mathsf{Exp} \to \mathscr{P}(\mathit{Conf})$ computes the set of reachable machine configurations for a given program:
$$\mathscr{E}(e) = \{c : \mathscr{I}(e) \Rightarrow^* c\}.$$
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:

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

Non-tail calls push a frame onto the stack and evaluate the call:

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

Function return pops a stack frame:

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

4 An infinite-state 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 $\hat{\mathscr{I}} : \mathsf{Exp} \to \widehat{Conf}$ pairs an expression with an empty environment, an empty store and an empty stack to create the initial abstract

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

```
-- Abstract state-space:
type AConf  = (Exp,AEnv,AStore,AKont)
type AEnv   = Var :-> AAddr
type AStore = AAddr :-> AD
type AD     =  (AVal)
data AVal   = AClo (Lambda, AEnv)
type AKont  = [AFrame]
type AFrame = (Var,Exp,AEnv)

data AAddr = ABind Var AContext
type AContext = [Call]
```

Fig. 4. The abstract configuration-space and its transliteration into Haskell. In the Haskell code, we defined abstract addresses to be able to support *k*-CFA-style polyvariance.

configuration:

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

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

$$\hat{\mathscr{A}}(lam, \hat{\rho}, \hat{\sigma}) = \{(lam, \hat{\rho})\} \qquad \text{[closure creation]}$$
$$\hat{\mathscr{A}}(v, \hat{\rho}, \hat{\sigma}) = \hat{\sigma}(\hat{\rho}(v)) \qquad \text{[variable look-up].}$$

The corresponding implementation is Haskell the mathematical version:

```
aeval :: (AExp,AEnv,AStore) -> AD
aeval (Ref v, ρ, σ) = σ!!(ρ!v)
aeval (Lam l, ρ, σ) = set $ AClo (l, ρ)
```

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

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

Because there are an infinite number of abstract configurations, a naïve implementation of this function may not terminate. Pushdown analysis provides a way of precisely computing this set and both finitely and compactly representing the result.

Transition relation The abstract transition relation $(\leadsto) \subseteq \widehat{Conf} \times \widehat{Conf}$ has three rules, one of which has become nondeterministic. In Haskell, we encode it as a function that returns lists of states:

```
astep :: AConf -> [AConf]
```

A tail call may fork because there could be multiple abstract closures that it is invoking:

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

In Haskell:

```
astep (App (f :@ ae), ρ, σ, κ) = [(e, ρ'', σ', κ) |
   AClo(v :=> e, ρ') <- Set.toList $ aeval(f, ρ, σ),
   let a = aalloc(v, App (f :@ ae)),
   let ρ'' = ρ' // [v ==> a],
   let σ' = σ  [a ==> aeval(ae, ρ, σ)] ]
```

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:

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

In Haskell:

```
astep (Let1 v call e, ρ, σ, κ) =
  [(App call, ρ, σ, (v, e, ρ) : κ)]
```

A function return pops a stack frame:

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

In Haskell:

```
astep (Ret ae, ρ, σ, (v, e, ρ') : κ) = [(e, ρ'', σ', κ)]
 where a = aalloc(v, Ret ae)
       ρ'' = ρ' // [v ==> a]
       σ' = σ  [a ==> aeval(ae, ρ, σ)]
```

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:

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

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:

$$(e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \sqsubseteq (e, \hat{\rho}', \hat{\sigma}', \hat{\kappa}') \text{ iff } \hat{\rho} \sqsubseteq \hat{\rho}' \text{ and } \hat{\sigma} \sqsubseteq \hat{\sigma}' \text{ and } \hat{\kappa} \sqsubseteq \hat{\kappa}'.$$

In Haskell, we define a typeclass for lattices:

```
class Lattice a where
 bot :: a
 top :: a
 (⊑) :: a -> a -> Bool
 (⊔) :: a -> a -> a
 (⊓) :: a -> a -> a
```

And, we can lift instances to sets and maps:

```
instance (Ord s, Eq s) => Lattice (ℙ s) where
 bot = Set.empty
 top = error "no representation of universal set"
 x ⊔ y = x ‘Set.union‘ y
 x ⊓ y = x ‘Set.intersection‘ y
 x ⊑ y = x ‘Set.isSubsetOf‘ y

instance (Ord k, Lattice v) => Lattice (k :-> v) where
 bot = Map.empty
 top = error "no representation of top map"
 f ⊑ g = Map.isSubmapOfBy (⊑) f g
 f ⊔ g = Map.unionWith (⊔) f g
 f ⊓ g = Map.intersectionWith (⊓) f g

(⊔) :: (Ord k, Lattice v) => (k :-> v) -> [(k,v)] -> (k :-> v)
f ⊔ [(k,v)] = Map.insertWith (⊔) k v f

(!!) :: (Ord k, Lattice v) => (k :-> v) -> k -> v
f !! k = Map.findWithDefault bot k f
```

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 $\alpha(c) \sqsubseteq \hat{c}$ and $c \Rightarrow c'$, then there exists $\hat{c}' \in \widehat{Conf}$ such that $\alpha(c') \sqsubseteq \hat{c}'$ and $\hat{c} \leadsto \hat{c}'$.

Proof
The proof follows by case analysis on the expression in the configuration. It is a straightforward adaptation of similar proofs, such as that of (Might 2007) for k-CFA. □

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

Fig. 5. $\widehat{\mathscr{PDA}} : \mathsf{Exp}\to\mathbb{PDA}$.

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 $([\![\texttt{(}f\ \text{æ}\texttt{)}]\!],\hat{\rho},\hat{\sigma},\hat{\kappa})$ in which $(\mathit{lam},\_)\in\hat{\mathscr{A}}(f,\hat{\rho},\hat{\sigma})$. 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 5 defines the program-to-PDA conversion function $\widehat{\mathscr{PDA}} : \mathsf{Exp}\to\mathbb{PDA}$. (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:

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

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 $\hat{S}$:

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

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:

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

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 (2004b); 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:

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

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:

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

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:

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

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

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

and the set E contains reachable edges:

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

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.

In this case, looking at the Haskell definition of a DSG helps:

[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.

```
type DSG control frame = (Edges control frame, control)
type Edges control frame = control :->  (StackAct frame,control)
```

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 $\widehat{\mathscr{RPDS}} : \mathsf{Exp} \to \mathbb{RPDS}$:

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

At this point, we must clarify how to embed the abstract transition relation into a pushdown transition relation:

```
adelta :: TopDelta AControl AFrame
adelta (e, ρ, σ)  = [ ((e', ρ', σ'), g) |
 (e', ρ', σ', κ) <- astep (e, ρ, σ, []),
 let g = case κ of
          []        -> Pop γ
          [γ1 , _ ] -> Push γ1
          [ _ ]     -> Unch ]

adelta' :: NopDelta AControl AFrame
adelta' (e, ρ, σ) = [ ((e', ρ', σ'), g) |
 (e', ρ', σ', κ) <- astep (e, ρ, σ, []),
 let g = case κ of
          [γ1]  -> Push γ1
          [ ]   -> Unch ]
```

7 Compacting a rooted pushdown system into a Dyck state graph

We now turn our attention to compacting a rooted pushdown system (defined intensionally) into a Dyck state graph (defined extensionally). That is, we want to find an implementation of the function $\mathscr{DSG}$. 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 $\mathscr{DSG}$. In the next section, we look at an optimized work-list driven algorithm that avoids the inefficiencies of this version.

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

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

Given a rooted pushdown system M, each application of the function $\mathscr{F}(M)$ 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
$\mathscr{DSG}(M) = \text{lfp}(\mathscr{F}(M))$.

Proof
Let $M = (Q, \Gamma, \delta, q_0)$. Let $f = \mathscr{F}(M)$. 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, $\mathscr{DSG}(M) \supseteq \text{lfp}(\mathscr{F}(M))$.

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

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 2004b).

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

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

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.

Moreover, we have carefully phrased the complexity in terms of "reachable" control states because, in practice, Dyck state graphs will be extremely sparse, and because, the maximum number of control states is exponential in the size of the input program. After the subsequent refinement, we will be able to develop a hierarchy of pushdown control-flow analyses that employs widening to achieve a polynomial-time algorithm at its foundation.

8 An efficient algorithm: 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 $q \rightarrowtail^{\varepsilon} q'$ into the following graph:

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

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:

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

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 $G_\varepsilon = (N, H)$, 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 $G_\varepsilon = (N, H)$:

$$\overleftarrow{G}_\varepsilon[s] = \left\{s' : (s', s) \in H\right\} \qquad \text{[ancestors]}$$
$$\overrightarrow{G}_\varepsilon[s] = \left\{s' : (s, s') \in H\right\} \qquad \text{[descendants].}$$

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.[5] 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 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 $\mathscr{F}' : \mathbb{RPDS} \to (\mathit{IDSG} \to \mathit{IDSG})$ generates the required iteration function (Figure 6). 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.

In Haskell, the function `dsg` will invoke the fixed point solver:

[5] 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.

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

Fig. 6. The fixed point of the function $\mathscr{F}'(M)$ contains the Dyck state graph of the rooted pushdown system M.

```
dsg :: (Ord control, Ord frame) =>
       (Delta control frame) ->
       control ->
       frame ->
       DSG control frame
dsg (δ,δ') q0 0 =
 (summarize (δ,δ') etg1 ecg1 [] dE dH, q0) where
 etg1 = (Map.empty // [q0 ==> Set.empty],
         Map.empty // [q0 ==> Set.empty])
 ecg1 = (Map.empty // [q0 ==> set q0],
         Map.empty // [q0 ==> set q0])
 (dE,dH) = sprout (δ,δ') q0
```

Figure 7 provides the Haskell code for `summarize`, which conducts the fixed point calculation, the executable equivalent of Figure 6:

```
summarize :: (Ord control, Ord frame) =>
             (Delta control frame) ->
             (ETG control frame) ->
             (ECG control) ->
             [control] ->
             [Edge control frame] ->
             [EpsEdge control] ->
             (Edges control frame)
```

To expose the strucutre of the computation, we've added a few types:

```
-- A set of edges, encoded as a map:
type Edges control frame =
 control :-> ℙ (StackAct frame,control)

-- Epsilon edges:
type EpsEdge control = (control,control)

-- Explicit transition graph:
type ETG control frame =
 (Edges control frame, Edges control frame)

-- Epsilon closure graph:
type ECG control =
 (control :-> ℙ(control), control :-> ℙ(control))
```

An explicit transition graph is an explicit encoding of the reachable subset of the transition relation. The function `summarize` takes six parameters:

1. the pushdown transition function;
2. the current explicit transition graph;
3. the current ε-closure graph;
4. a work-list of states to add;
5. a work-list of explicit transition edges to add; and
6. a work-list of ε-closure transition edges to add.

The function `summarize` processes ε-closure edges first, then explicit transition edges and then individual states. It *must* process ε-closure edges first to ensure that the ε-closure graph is closed when considering the implications of other edges.

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:

$$sprout_{(Q,\Gamma,\delta)} : Q \to (\mathscr{P}(\delta) \times \mathscr{P}(Q \times Q)),$$

```
summarize (δ,δ') (fw,bw) (fe,be) [] [] [] = fw

summarize (δ,δ') (fw,bw) (fe,be) (q:dS) [] []
 | fe ‘contains‘ q = summarize (δ,δ') (fw,bw) (fe,be) dS [] []
summarize (δ,δ') (fw,bw) (fe,be) (q:dS) [] [] =
 summarize (δ,δ') (fw',bw') (fe',be') dS dE' dH' where
  (dE',dH') = sprout (δ,δ') q
  fw' = fw ⊔ [q ==> Set.empty]
  bw' = bw ⊔ [q ==> Set.empty]
  fe' = fe ⊔ [q ==> set q]
  be' = be ⊔ [q ==> set q]

summarize (δ,δ') (fw,bw) (fe,be) dS ((q,g,q'):dE) []
 | (q,g,q') ‘isin'‘ fw  = summarize (δ,δ') (fw,bw) (fe,be) dS dE []
summarize (δ,δ') (fw,bw) (fe,be) dS ((q,Push ,q'):dE) [] =
 summarize (δ,δ') (fw',bw') (fe',be') dS' dE'' dH' where
  (dE',dH') = addPush (fw,bw) (fe,be) (δ,δ') (q,Push ,q')
  dE'' = dE' ++ dE''
  dS' = q':dS
  fw' = fw ⊔ [q  ==> set (Push ,q')]
  bw' = bw ⊔ [q' ==> set (Push ,q) ]
  fe' = fe ⊔ [q  ==> set q ]
  be' = fe ⊔ [q' ==> set q']

summarize (δ,δ') (fw,bw) (fe,be) dS ((q,Pop ,q'):dE) [] =
 summarize (δ,δ') (fw',bw') (fe',be') dS' dE'' dH' where
  (dE',dH') = addPop (fw,bw) (fe,be) (δ,δ') (q,Pop ,q')
  dE'' = dE ++ dE'
  dS' = q':dS
  fw' = fw ⊔ [q  ==> set (Pop ,q')]
  bw' = bw ⊔ [q' ==> set (Pop ,q) ]
  fe' = fe ⊔ [q  ==> set q ]
  be' = fe ⊔ [q' ==> set q']

summarize (δ,δ') (fw,bw) (fe,be) dS ((q,Unch,q'):dE) [] =
 summarize (δ,δ') (fw',bw') (fe',be') dS' dE [(q,q')] where
  dS' = q':dS
  fw' = fw ⊔ [q  ==> set (Unch,q')]
  bw' = bw ⊔ [q' ==> set (Unch,q) ]
  fe' = fe ⊔ [q  ==> set q ]
  be' = fe ⊔ [q' ==> set q']

summarize (δ,δ') (fw,bw) (fe,be) dS dE ((q,q'):dH)
 | (q,q') ‘isin‘ fe  = summarize (δ,δ') (fw,bw) (fe,be) dS dE dH
summarize (δ,δ') (fw,bw) (fe,be) dS dE ((q,q'):dH) =
 summarize (δ,δ') (fw,bw) (fe',be') dS dE' dH' where
   (dE',dH') = addEmpty (fw,bw) (fe,be) (δ,δ') (q,q')
   fe' = fe ⊔ [q  ==> set q ]
   be' = fe ⊔ [q' ==> set q']
```

Fig. 7. A Haskell implementation of pushdown control-state reachability.

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

$$\begin{tikzcd} & \boxed{s} \arrow[dl, dotted, "\delta"', "\varepsilon"] \arrow[dr, dotted, "\gamma_+", "\delta"'] & \\ q' & & q'' \end{tikzcd}$$

which means if adding control state s:

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

Formally:

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

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

In Haskell:

```
sprout :: (Ord control) =>
          Delta control frame ->
          control ->
          ([Edge control frame], [EpsEdge control])
sprout (δ,δ’) q = (dE, dH) where
  edges = δ’ q
  dE = [ (q,g,q’) | (q’,g) <- edges, isPush g ]
  dH = [ (q,q’)   | (q’,g) <- edges, isUnch g ]
```

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:

$$addPush_{(Q,\Gamma,\delta)} : \mathbb{DSG} \times \mathbb{ECG} \to \delta \to (\mathscr{P}(\delta) \times \mathscr{P}(Q \times Q)),$$

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

$$\begin{tikzcd} \boxed{s} \arrow[r, "\gamma_+"] \arrow[rrr, dotted, bend right, "\varepsilon", "\varepsilon"'] & \boxed{q} \arrow[r, "\varepsilon", "\varepsilon"'] & q' \arrow[r, dotted, "\gamma_-", "\delta"'] & q'' \end{tikzcd}$$

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:

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

In Haskell:

```
addPush :: (Ord control) =>
           ETG control frame ->
           ECG control ->
           Delta control frame ->
           Edge control frame ->
           ([Edge control frame], [EpsEdge control])
addPush (fw,bw) (fe,be) (δ,δ') (s,Push γ,q) = (dE,dH) where
 qset' = Set.toList $ fe!q
 dE = [ (q',g,q'') | q' <- qset', (q'',g) <- δ q' γ, isPop g ]
 dH = [ (s,q'')    | (q',Pop _,q'') <- dE ]
```

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:

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

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

$$\begin{tikzcd}
s \arrow[r, "\gamma_+"] \arrow[rrr, bend right, dotted, "\varepsilon", "\varepsilon"'] & s' \arrow[r, "\varepsilon", "\varepsilon"'] & \boxed{s''} \arrow[r, "\gamma_-", "\delta"'] & \boxed{q}
\end{tikzcd}$$

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:

$$
\begin{aligned}
&\mathit{addPop}_{(Q,\Gamma,\delta)}(G, G_\varepsilon)(s'' \overset{\gamma_-}{\rightarrowtail} q) = (\Delta E, \Delta H), \text{ where}\\
&\Delta E = \emptyset \text{ and } \Delta H = \left\{ s \rightarrowtail q : s' \in \overleftarrow{G}_\varepsilon[s''] \text{ and } s \overset{\gamma_+}{\rightarrowtail} s' \in G \right\}.
\end{aligned}
$$

In Haskell:

```
addPop :: (Ord control) =>
          ETG control frame ->
          ECG control ->
          Delta control frame ->
          Edge control frame ->
          ([Edge control frame], [EpsEdge control])
addPop (fw,bw) (fe,be) (δ,δ') (s'',Pop γ,q) = (dE,dH) where
 sset' = Set.toList $ be!s''
 dH = [ (s,q) | s' <- sset',
                (g,s) <- Set.toList $ bw!s', isPush g ]
 dE = []
```

Clearly, we could eliminate the new edges parameter dE for the function addPop, but we have retained it for stylistic symmetry.

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 *addEmpty* function:

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

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

$$\begin{tikzcd}
s \arrow[r, "\gamma_+"] \arrow[rrrrr, dotted, bend right=60, "\varepsilon", "\varepsilon"'] & s' \arrow[r, "\varepsilon", "\varepsilon"'] \arrow[rr, dotted, bend left=50, "\varepsilon", "\varepsilon"'] \arrow[rrr, dotted, bend right=40, "\varepsilon", "\varepsilon"'] & \boxed{s''} \arrow[r, "\varepsilon"] \arrow[rr, dotted, bend left=50, "\varepsilon", "\varepsilon"'] & \boxed{s'''} \arrow[r, "\varepsilon", "\varepsilon"'] & s'''' \arrow[r, dotted, "\gamma_-", "\delta"'] & q
\end{tikzcd}$$

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:

$$\begin{aligned}
&\mathit{addEmpty}_{(Q,\Gamma,\delta)}(G,G_\varepsilon)(s'' \rightarrowtail s''') = (\Delta E, \Delta H), \text{ where}\\
&\Delta E = \left\{s'''' \overset{\gamma_-}{\rightarrowtail} q : s' \in \overleftarrow{G}_\varepsilon[s''] \text{ and } s'''' \in \overrightarrow{G}_\varepsilon[s'''] \text{ and } s \overset{\gamma_+}{\rightarrowtail} s' \in G\right\}\\
&\Delta H = \left\{s \rightarrowtail q : s' \in \overleftarrow{G}_\varepsilon[s''] \text{ and } s'''' \in \overrightarrow{G}_\varepsilon[s'''] \text{ and } s \overset{\gamma_+}{\rightarrowtail} s' \in G\right\}\\
&\quad \cup \left\{s' \rightarrowtail s''' : s' \in \overleftarrow{G}_\varepsilon[s'']\right\}\\
&\quad \cup \left\{s'' \rightarrowtail s'''' : s'''' \in \overrightarrow{G}_\varepsilon[s''']\right\}\\
&\quad \cup \left\{s' \rightarrowtail s'''' : s' \in \overleftarrow{G}_\varepsilon[s''] \text{ and } s'''' \in \overrightarrow{G}_\varepsilon[s''']\right\}.
\end{aligned}$$

In Haskell, the function `addEmpty` has many cases to consider:

```
addEmpty :: (Ord control) =>
            ETG control frame ->
            ECG control ->
            Delta control frame ->
            EpsEdge control ->
            ([Edge control frame], [EpsEdge control])
addEmpty (fw,bw) (fe,be) (δ,δ') (s'',s''') = (dE,dH) where
 sset'    = Set.toList $ be!s''
 sset'''' = Set.toList $ fe!s'''
 dH'   = [ (s',s'''')  | s' <- sset', s'''' <- sset'''' ]
 dH''  = [ (s',s''')   | s' <- sset' ]
 dH''' = [ (s'',s'''') | s'''' <- sset'''' ]

 sEdges = [ (g,s) | s' <- sset', (g,s) <- Set.toList $ bw!s' ]

 dE = [ (s'''',g',q) | s'''' <- sset'''',
                       (g,s) <- sEdges,
                       isPush g, let Push γ = g,
                       (q,g') <- δ s'''' γ,
                       isPop g' ]

 dH'''' = [ (s,q) | (_,s) <- sEdges, (_,_,q) <- dE ]

 dH = dH' ++ dH'' ++ dH''' ++ dH''''
```

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 $\mathscr{F}'(M)(G, G_\varepsilon, \Delta G) = (G', G'_\varepsilon, \Delta G')$, 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
$\text{lfp}(\mathscr{F}'(M)) = (\mathscr{DSG}(M), G_\varepsilon, (\emptyset, \emptyset), \emptyset)$.

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:

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

We can redefine the concrete evaluation function:

$$\begin{aligned} &\mathscr{E}(e) = \mathrm{lfp}(f_e), \text{ where } f_e : \mathit{System} \to \mathit{System} \text{ and} \\ &f_e(C) = \{\mathscr{I}(e)\} \cup \{c' : c \in C \text{ and } c \Rightarrow c'\}. \end{aligned}$$

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:

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

and then the abstract evaluation function:

$$\begin{aligned} &\hat{\mathscr{E}}(e) = \mathrm{lfp}(\hat{f}_e), \text{ where } \hat{f}_e : \widehat{\mathit{System}} \to \widehat{\mathit{System}} \text{ and} \\ &\hat{f}_e(\hat{C}) = \left\{\hat{\mathscr{I}}(e)\right\} \cup \left\{\hat{c}' : \hat{c} \in \hat{C} \text{ and } \hat{c} \leadsto \hat{c}'\right\}. \end{aligned}$$

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:

$$\begin{aligned} \widehat{\mathit{System}}' &= \mathscr{P}\left(\widehat{\mathit{PConf}}\right) \times \widehat{\mathit{Store}} \\ \hat{\pi} \in \widehat{\mathit{PConf}} &= \mathsf{Exp} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Kont}}. \end{aligned}$$

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

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

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

$$\begin{aligned}
\hat{f}'_e(\hat{P}, \hat{\sigma}) &= (\hat{P}', \hat{\sigma}'), \text{ where} \\
\hat{P}' &= \left\{ \hat{\pi}' : \hat{\pi} \overset{\hat{\sigma}}{\rightharpoonup} (\hat{\pi}', \hat{\sigma}'') \right\} \cup \{\hat{\pi}_0\} \\
\hat{\sigma}' &= \bigsqcup \left\{ \hat{\sigma}'' : \hat{\pi} \overset{\hat{\sigma}}{\rightharpoonup} (\hat{\pi}', \hat{\sigma}'') \right\} \\
(\hat{\pi}_0, \langle\rangle) &= \hat{\mathscr{I}}(e).
\end{aligned}$$

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:

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

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.[6]

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:

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

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 $(\overset{\hat{\sigma}}{\underset{g}{\rightarrowtail}}) \subseteq \widehat{PState} \times \widehat{PState} \times Store$ determines if a transition is possible under the specified store and stack-action:

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

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

$$(e,\hat{\rho}) \overset{\hat{\sigma}}{\underset{\varepsilon}{\rightarrowtail}} ((e',\hat{\rho}'),\hat{\sigma}') \text{ iff } (e,\hat{\rho},\hat{\sigma},\hat{\kappa}) \leadsto (e',\hat{\rho}',\hat{\sigma}',\hat{\kappa}').$$

Now, we can redefine the iteration function (Figure 8).

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. □

[6] 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.*, $|\widehat{Frame}| = |\mathsf{Var}|$.

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

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

Fig. 8. An ε-closure graph-powered iteration function for pushdown control-flow analysis with a single-threaded store.

10 Introspection for abstract garbage collection

Abstract garbage collection (Might and Shivers 2006b) 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 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 a full traversal of the stack 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:

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

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 $\mathit{Reachable} : \widehat{Conf} \to \mathscr{P}(\widehat{Addr})$ first computes the root set, and then the transitive closure of an address-to-address adjacency relation:

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

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

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

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

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

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

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

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:

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

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

10.1 Garbage collection in monotonic introspective pushdown systems

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

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

11 Problem: Reachability for introspective pushdown systems is uncomputable

As currently formulated, computing control-state reachability for introspective pushdown systems is uncomputable. The problem is that the transition relation expects to enumerate every possible stack for every control point at every transition. Because there are an unbounded number of stacks at each control point, it is impossible to know (without peering into the otherwise-opaque contents of the transition relation) all the successors.

To make introspective pushdown systems computable, we must first refine our definition of introspective pushdown systems to operate on *sets* of stacks and to include a monotonicity constraint.

A **monotonic 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. $\delta \subseteq Q \times \mathscr{P}(\Gamma^*) \times \Gamma_\pm \times Q$ is a monotonic transition relation; and
4. q_0 is a distinguished root control state.

The monotonicity constraint on the transition relations guarantees that as the set of stacks grows larger, no previously included transitions will suddenly be excluded:

$$T \subseteq T' \text{ and } \delta(q,T,g,q') \text{ implies } \delta(q,T',g,q').$$

11.1 Garbage collection in monotonic introspective pushdown systems

Of course, we must adapt abstract garabge collection to this refined framework. To convert the garbage-collecting, abstracted CESK machine into a monotonic introspective pushdown system, we use the function $\widehat{\mathscr{IPDS}}'$: $\mathsf{Exp} \to \mathbb{IPDS}$:

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

Assuming we can overcome the difficulty of computing with an opaquely represented set of stacks, we can already see that this control-state reachability with garbage collection in this formulation should be computable: garbage collection operates on sets of frames, and for any given control point there a finite number of sets of sets[7] of frames.

The last challenge to consider before we can delve into the mechanics of computing reachable control states is *how* to represent the sets of stacks that may be paired with each control state. Fortunately, a regular language can describe the set of stacks at a control point, *and*, fortuitously, this regular language is already encoded in the structure of the Dyck state graph that we will accumulate while computing reachable control states. As we develop an algorithm for control-state reachability, we will exploit this insight.

12 Computing reachability for monotonic introspective pushdown systems

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

As with ordinary pushdown systems, 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.

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, $\mathscr{DSG} : \mathbb{IPDS} \to \mathbb{DSG}$.

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

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

and the set E contains reachable edges:

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

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

[7] This is not a typo.

12.1 Compiling to Dyck state graphs

We now turn our attention to compiling an monotonic introspective pushdown system (defined implicitly) into a Dyck state graph (defined explicitly). That is, we want an implementation of the function $\mathscr{DSG}$. As with ordinary pushdown systems, 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 $\mathscr{DSG}$.

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

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

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

12.2 Computing a round of $\mathscr{F}$

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:

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

Figure 9 renders this DSG to NFA converter in Haskell.

In Haskell, we must also change the definition of transition functions to accept an NFA describing all stacks:

```
type IDelta control frame =
  (ITopDelta control frame, INopDelta control frame)

type ITopDelta control frame =
  control ->
  NFA control frame ->
  frame ->
  [(control,StackAct frame)]
```

```
stacks :: (Ord control, Ord frame) =>
          ETG control frame ->
          ECG control ->
          control ->
          control ->
          NFA control frame
stacks (fw,bw) (fe,be) s0 s' = (f'',b'',s0,s') where

 fedges = [ (q,Set.fromAscList trans') |
            (q,trans) <- Map.toAscList fw,
            let trans' = [ (Just c,q') |
                           (g,q') <- Set.toAscList $ trans,
                           isPush g,
                           let c = frame g ] ]

 fedges' = [ (q,Set.fromAscList trans') |
             (q,trans) <- Map.toAscList fe,
             let trans' = [ (Nothing,q') |
                            q' <- Set.toList $ trans ] ]

 f  = Map.fromAscList fedges
 f' = Map.fromAscList fedges'
 f'' = Map.union f f'

 bedges = [ (q,Set.fromAscList trans') |
            (q,trans) <- Map.toAscList bw,
            let trans' = [ (Just c,q') |
                           (g,q') <- Set.toAscList $ trans,
                           isPush g,
                           let c = frame g ] ]

 bedges' = [ (q,Set.fromAscList trans') |
             (q,trans) <- Map.toAscList be,
             let trans' = [ (Nothing,q') |
                            q' <- Set.toList $ trans ] ]

 b  = Map.fromAscList bedges
 b' = Map.fromAscList bedges'
 b'' = Map.union b b'
```

Fig. 9. A rendering in Haskell of the conversion from Dyck state graphs to NFAs describing realizable stacks.

```
type INopDelta control frame =
  control ->
  NFA control frame ->
  [(control,StackAct frame)]
```

This means that any function that invokes the pushdown transition relation must have access to either an NFA describing all stacks for the initial point, or it must have enough information to compute it. In the original formulation, there are four functions that require this—`sprout`, `addPush`, `addPop` and `addEmpty`:

```
isprout :: (Ord control) =>
           IDelta control frame ->
           control ->
           NFA control frame ->
           ([Edge control frame], [EpsEdge control])

addIPush :: (Ord control, Ord frame) =>
            control ->
            ETG control frame ->
            ECG control ->
            IDelta control frame ->
            Edge control frame ->
            ([Edge control frame], [EpsEdge control])

addIPop :: (Ord control, Ord frame) =>
           control ->
           ETG control frame ->
           ECG control ->
           IDelta control frame ->
           Edge control frame ->
           ([Edge control frame], [EpsEdge control])

addIEmpty :: (Ord control, Ord frame) =>
             control ->
             ETG control frame ->
             ECG control ->
             IDelta control frame ->
             EpsEdge control ->
             ([Edge control frame], [EpsEdge control])
```

For sprouting, we pass the exact NFA, since we're concerned with only one control state. For the remainder, we pass the initial control state, so that it may compute the NFA as necessary.

12.3 Correctness

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

Theorem 12.1
$\mathscr{DSG}(M) = \text{lfp}(\mathscr{F}(M))$.

Proof
Let $M = (Q, \Gamma, \delta, q_0)$. Let $f = \mathscr{F}(M)$. 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, $\mathscr{DSG}(M) \supseteq \text{lfp}(\mathscr{F}(M))$.

To show $\mathscr{DSG}(M) \subseteq \text{lfp}(\mathscr{F}(M))$, suppose this is not the case. Then, there must be at least one edge in $\mathscr{DSG}(M)$ that is not in $\text{lfp}(\mathscr{F}(M))$. By the defintion of $\mathscr{DSG}(M)$, 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 $\text{lfp}(\mathscr{F}(M)$. Because the proceeding edge is in $\text{lfp}(\mathscr{F}(M))$, the state s *is* in $\text{lfp}(\mathscr{F}(M))$. 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 $\text{lfp}(\mathscr{F}(M))$, which is a contradiction. Hence, $\mathscr{DSG}(M) \subseteq \text{lfp}(\mathscr{F}(M))$. $\square$

12.4 Simplifying garbage collection in introspective pushdown systems

Because monotonic introspective pushdown systems consider all possible stacks at every control point, they may provide more path-sensitivity than necessary for the intended application. This is certainly the case with abstract garbage collection.

At the very least, abstract garbage collection is indifferent to the ordering of stack frames: sets of frames suffice. Even then, an analyzer need not consider all sets of sets of frames. For instance, if the regular expression $(A|B)C^*$ describes all the stacks at a given control point, then the sets of possible frames are $\{A\}$, $\{B\}$, $\{A,C\}$ and $\{B,C\}$. Because $\{A\} \subseteq \{A,C\}$ and $\{B\} \subseteq \{B,C\}$, the monotonicity of the abstract transition relation guarantees that any states considered by collecting $\{A\}$ or $\{B\}$ would be covered by states considered when collecting $\{A,C\}$ or $\{B,C\}$. This simplification comes at not cost to precision.

If one is willing to sacrifice precision, then one can union all of the sets of reachable frames together when garbage collecting at a given control point. In the prior example, it would mean considering garbage collection with only the set of frames $\{A,B,C\}$.

13 An algorithm for introspective pushdown analysis with garbage collection

The reachability-based analysis for a pushdown system described in Section 7 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, and we presented an efficient algorithm to compute those in Section 8. 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?

The crucial addition to the algorithm is maintaining for each node q' in the DSG a set of ε*-predecessors*, i.e., nodes q, such that $q \longmapsto\!\!\!\rightarrow_{M}^{\vec{g}} q'$ and $[\vec{g}] = \varepsilon$. 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.

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 $q \xrightarrow{\varepsilon} q'$ 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.

14 Experimental evaluation

A fair comparison between different families of analyses should compare both precision and speed. We have implemented k-CFA for a subset of R5RS Scheme and instrumented it with a possibility 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 garbage 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 (Might et al. 2007).

14.1 Plain k-CFA vs. pushdown k-CFA

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 10). Most benchmarks are taken from the CFA literature: `mj09` is a running example

Program	#*e*	#*v*	*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	32	32	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	34	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	> 32662	> 88548	–	1761	4046	2	53	52	2	53	52	2
`blur`	40	20	0	4686	7606	4	115	146	4	90	95	10	68	76	10
			1	123	149	10	94	101	10	76	82	10	75	81	10
`loop2`	41	14	0	149	163	7	69	73	7	43	46	7	34	35	7
			1	> 10867	> 16040	–	411	525	3	151	163	3	145	156	3
`sat`	51	23	0	3844	5547	4	545	773	4	1137	1543	4	254	317	4
			1	> 28432	> 37391	–	12828	16846	4	958	1314	5	71	73	10

Fig. 10. Benchmark results for toy programs. 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 of the plain *k*-CFA denote the case when the analysis explored more than 10^5 *configurations* (*i.e.*, control statce coupled with continuations) or did not finish within 30 minutes. For such cases we do not report on singleton variables.

from the work of Midtgaard and Jensen designed to exhibit a non-trivial return-flow behavior (Midtgaard 2007), `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 (Van Horn and Mairson 2008), `loop2` is an example from the Might's dissertation that was used to demonstrate the impact of abstract GC (Might 2007, Section 13.3), `sat` is a brute-force SAT-solver with backtracking.

14.1.1 *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.

Running on the benchmarks, we have revalidated hypotheses about the improvements to precision granted by both pushdown analysis (Vardoulakis and Shivers 2010) and abstract garbage collection (Might 2007). The table in Figure 10 contains our detailed results on the precision of the analysis. In order to make the comparison fair, in the table we report on the numbers of *control states*, which do not contain a stack components and are the nodes

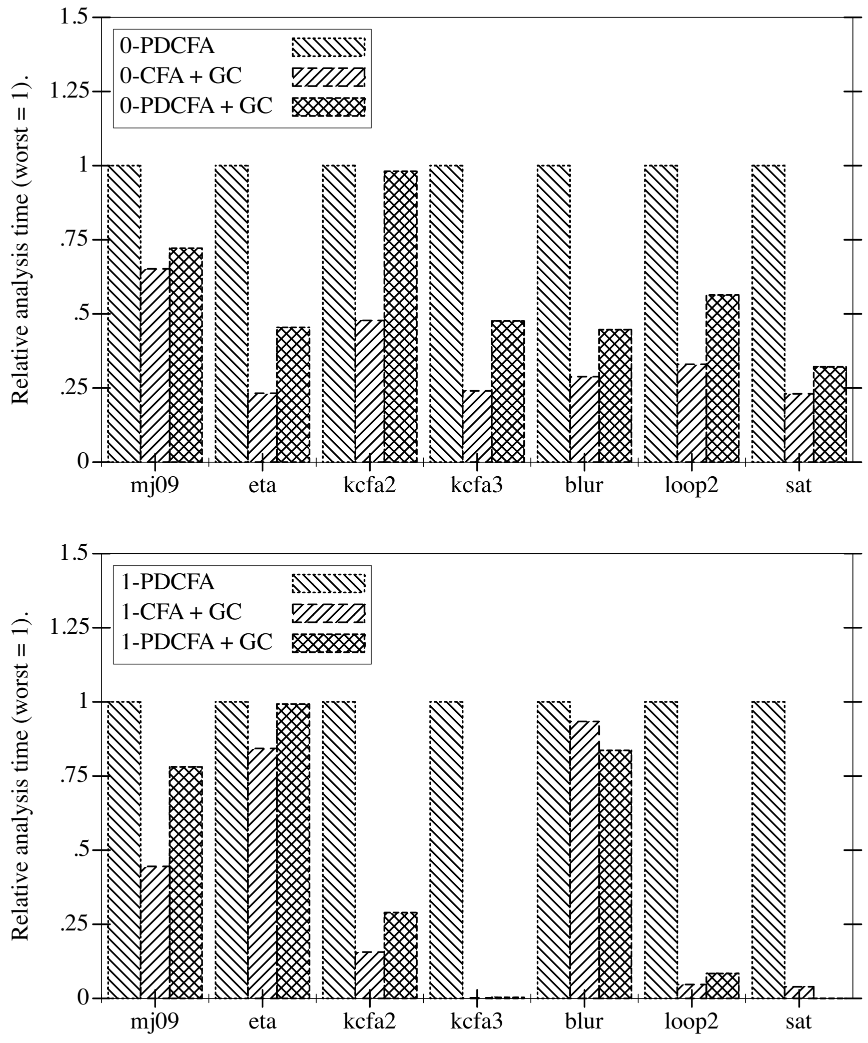


Fig. 11. Analysis times relative to worst (= 1) in class; smaller is better. On the left is the monovariant 0CFA class of analyses, on the right is the polyvariant 1CFA class of analyses. (Non-GC k-CFA omitted.)

of the constructed DSG. In the case of plain k-CFA, control states are coupled with stack pointers to obtain *configurations*, whose resulting number is significantly bigger.

The SAT-solving benchmark 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.

14.1.2 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

Program	#e	#v	!#v	$k=0$, GC off			$k=0$, GC on			$k=1$, GC off			$k=1$, GC on		
`primtest`	155	44	16	790	955	14″	113	127	1″	>43146	>54679	∞	442	562	13″
`rsa`	211	93	36	1267	1507	23″	355	407	6″	20746	28895	21′	926	1166	28″
`regex`	344	150	44	943	956	54″	578	589	45″	1153	1179	88″	578	589	50″
`scm2java`	1135	460	63	376	375	13″	376	375	13″	376	375	14″	376	375	13″

Fig. 12. Benchmark results for real-life programs. The first four columns provide the name of a program, the number of expressions and variables in the program in the ANF, and the number of singleton variables revealed by the analysis (same in all cases). For each of four combinations of $k \in \{0,1\}$ and garbage collection on or off, the first two columns in a group show the number of visited control states and edges, respectively, and the third one shows absolute time of running the analysis (for both less is better). The results of the analyses are presented in minutes (′) or seconds (″), where ∞ stands for an analysis, which has been interrupted due to the an execution time greater than 30 minutes.

states (Vardoulakis and Shivers 2010, Section 6). There is a similar correlation in our fused analysis, but it is not as strong or as absolute. Since most of the programs from our toy suite run for less than a second, we do not report on the absolute time. Instead, the histogram on Figure 11 presents normalized relative times of analyses' executions. To our observation the pure machine-style k-CFA is always significantly worse in terms of execution time than either with GC or push-down system, so we excluded the plain, non-optimised k-CFA from the comparison.

Our earlier implementation of a garbage-collecting pushdown analysis (Earl et al. 2012) did not fully exploit the opportunities for caching ε-predecessors, as described in Section 13. This led to significant inefficiencies of the garbage-collecting analyser with respect to the mere one, even though the former one observed a smaller amount of states and in some cases found larger amounts of singleton variables. After this issue has been fixed, it became clearly visible that in all cases the GC-optimised analyser is strictly more fast than it non-optimised pushdown counterpart.

Although caching of ε-predecessors and ε-summary edges is relatively cheap, it is not free, since maintaining the caches requires some routine machinery at each iteration of the analyser. This explains the loss in performance of the garbage-collecting pushdown analysis with respect to the GC-optimised mere CFA.

As it follows from the plot, fused analysis is always faster than one the mere pushdown analysis, and about a fifth of the time, it is beats the plain CFA with garbage collection in terms of performance. When the fused analysis is slower than just a GC-optimized one, 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 analysis, the amortized penalty is a small and acceptable price to pay for improvements to precision.

14.2 Analysing real-life programs with garbage-collecting pushdown CFA

Even though our prototype implementation is just a proof of concept, we found it curious to check what would be the results of running a garbage-collecting version of pushdown analyser not on a suite of toy programs, tailored for particular functional programming patterns, but on a set of real-life programs. In order to set this experiment, we have chosen four programs, dealing with numeric and symbolic computations:

- `primtest` – an implementation of the probabilistic Fermat and Solovay-Strassen primality testing in Scheme for the purpose of large prime generation;
- `rsa` – an implementation of the RSA public-key cryptography;
- `regex` – an implementation of regular expression matcher in Scheme using Brzozovski derivatives (Might et al. 2011; Owens et al. 2009);
- `scm2java` – scm2java is a Scheme to Java compiler;

We ran our benchmark suite on a 2.7 GHz Intel Core i7 OS X machine with 8 Gb RAM. Unfortunately, the mere CFA timed out on most of these examples, so we had to exclude it from the comparison an focus on the effect of a pushdown analyser only. The table in Figure 12 presents the results of running the benchmarks for $k = 0, 1$ with a garbage collector on and off. Surprisingly, for each of the six program, those cases, which terminated within 30 minutes, delivered the same number of found singleton variables. However, the numbers of observed states and runtimes are indeed different in most of the cases except `scm2java`, for which all the four versions of the analysis were precise enough to actually evaluate the program. Time-wise, the results of the experiment demonstrate the general positive effect of the abstract garbage collection in a pushdown setting, which might improve the analysis performance for more than two orders of magnitude.

15 Discussion: Applications

Pushdown control-flow analysis offers more precise control-flow analysis results than the classical finite-state CFAs. Consequently, introspective 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 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 introspective pushdown control-flow analysis. (We leave an in-depth formulation and study of these analyses to future work.)

15.1 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.

15.2 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."

16 Related work

An earlier version of this work appeared in ICFP 2012 (Earl et al. 2012). The complete development of pushdown analysis from first principles stands as a new contribution, and it constitutes an alternative development of CFA2. It goes well beyond the ICFP 2012 work (and work on CFA2) by specifying specific mechanisms for reducing the complexity to polynomial time as well. An immediate advantage of the complete development is its exposure of parameters for controlling polyvariance and context-sensitivity. We also provide an unambiguous reference implementation of control-state reachability in Haskell. We felt this was necessary to shine a light on the "dark corners" in the formalism, and in fact, it helped expose both bugs and implicit design decisions that were reflected in the revamped text of this work. The development of introspective pushdown systems is also more complete and more rigorous. We expose the critical monotonicity constraint absent from the ICFP 2012 work, and we specify the implementation of control-state reachability for introspective pushdown systems in greater detail. More importantly, this work uses additional techniques to improve the performance of the implementation and discusses those changes.

Garbage-collecting 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 motivating work for our own is Vardoulakis and Shivers very recent discovery of 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 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.

Calculation approach to abstract interpretation Midtgaard and Jensen (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.* (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 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 (Bouajjani et al. 1997; Kodumal and Aiken 2004a; 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 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 (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. Finally, our technique is not restricted to typed programs.

Model-checking higher-order recursion schemes There is terminology overlap with work by Kobayashi (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 (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. In particular, higher-order recursions schemes are limited to model-checking programs in the simply-typed lambda-calculus with recursion.

17 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 the anonymous reviewers of ICFP 2012 for their detailed and helpful reviews. 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

Bouajjani, A., J. Esparza, and O. Maler (1997). Reachability analysis of pushdown automata: Application to Model-Checking. In *Proceedings of the 8th International Conference on Concurrency Theory*, CONCUR '97, pp. 135–150. Springer-Verlag.

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

Cousot, P. and R. Cousot (1977). 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, pp. 238–252. ACM Press.

Earl, C., I. Sergey, M. Might, and D. Van Horn (2012). Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pp. 177–188. ACM.

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

Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen (1993, June). The essence of compiling with continuations. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 237–247. ACM.

Kobayashi, N. (2009, January). Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Volume 44 of *POPL '09*, pp. 416–428. ACM.

Kodumal, J. and A. Aiken (2004a, June). The set constraint/CFL reachability connection in practice. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pp. 207–218. ACM.

Kodumal, J. and A. Aiken (2004b, June). The set constraint/CFL reachability connection in practice. *SIGPLAN Not. 39*, 207–218.

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

Midtgaard, J. (2007). *Transformation, Analysis, and Interpretation of Higher-Order Procedural Programs*. Ph. D. thesis, University of Aarhus.

Midtgaard, J. and T. P. Jensen (2009). 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*, pp. 287–298. ACM.

Might, M. (2007, June). *Environment Analysis of Higher-Order Languages*. Ph. D. thesis, Georgia Institute of Technology.

Might, M., B. Chambers, and O. Shivers (2007, January). Model checking via Gamma-CFA. In *Verification, Model Checking, and Abstract Interpretation*, pp. 59–73.

Might, M., D. Darais, and D. Spiewak (2011). Parsing with derivatives: a functional pearl. In *ICFP '11: Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming*, pp. 189–195. ACM.

Might, M. and T. Prabhu (2009). Interprocedural dependence analysis of higher-order programs via stack reachability. In *Proceedings of the 2009 Workshop on Scheme and Functional Programming*.

Might, M. and O. Shivers (2006a). Environment analysis via Delta-CFA. In *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 127–140. ACM.

Might, M. and O. Shivers (2006b). 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*, pp. 13–25. ACM.

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

Owens, S., J. Reppy, and A. Turon (2009). Regular-expression derivatives re-examined. *Journal of Functional Programming 19*(02), 173–190.

Rehof, J. and M. Fähndrich (2001). 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*, pp. 54–66. ACM.

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

Reps, T., S. Schwoon, S. Jha, and D. Melski (2005, October). Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming 58*(1-2), 206–263.

Shivers, O. G. (1991). *Control-Flow Analysis of Higher-Order Languages*. Ph. D. thesis, Carnegie Mellon University.

Sipser, M. (2005, February). *Introduction to the Theory of Computation* (2 ed.). Course Technology.

Van Horn, D. and H. G. Mairson (2008). Deciding *k*CFA is complete for EXPTIME. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, pp. 275–282. ACM.

Vardoulakis, D. and O. Shivers (2010). Cfa2: a Context-Free Approach to Control-Flow Analysis. In *European Symposium on Programming (ESOP)*, Volume 6012 of *LNCS*, pp. 570–589. Springer.

Wright, A. K. and S. Jagannathan (1998, January). Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems 20*(1), 166–207.