Concrete Semantics for Pushdown Analysis: The Essence of Summarization

J. Ian Johnson and David Van Horn

Northeastern University
{ianj,dvanhorn}@ccs.neu.edu

Abstract. Pushdown analysis is better than finite-state analysis in precision and performance. Why then have we not seen total widespread adoption of these techniques? For one, the known techniques are technically burdened and difficult to understand or extend. Control structure of the programming language gets pulled into the model of computation, which makes extensions to non-pushdown control structures, such as `call/cc` or `shift` and `reset`, non-trivial.
We show a derivational approach to abstract interpretation that yields transparently sound static analyses that can precisely match calls and returns when applied to well-known abstract machines. We show that adding memoization and segmenting the continuation into bounded pieces leads to machines that abstract to static analyses for context-free reachability by simply bounding the stores. This technique allows us to derive existing, more technically involved analyses, and a novel pushdown analysis for delimited, composable control.

1 Introduction

Programs in higher-order languages heavily use function calls and method dispatch as part of their control flow. Until recently, flow analyses for higher-order languages could not handle return flow precisely [Vardoulakis and Shivers, 2011b, Earl et al., 2010], which leads to several spurious paths (and thus false positives) due to the pervasiveness of function/method calls and subsequent returns. These works, called CFA2 and PDCFA respectively, use pushdown automata as their approximation's target model of computation. They are hence called "pushdown analyses."[1] CFA2 and PDCFA have difficult details to easily apply to an off-the-shelf semantics — especially if they feature non-local control transfer that breaks the pushdown model.

There is a systematic process for transforming off-the-shelf programming language semantics into a form amenable to *regular* analysis that has been widely applied with great success to production programming languages Van Horn and Might [2010] (a technique called abstracting abstract machines, or AAM). A contribution of this paper is a systematic process to construct *pushdown* analyses of programming languages, due to the precision (and often performance) benefits.

[1] We will refer to the classic finite model analyses as "regular analyses" after regular languages.

Testing new ideas in analysis for improving precision or increasing performance can be a difficult venture. We contend that the machinery that we employ in the abstract should have a concrete counterpart that maintains the meaning of the language so that we can see their effect without introducing approximations which might mask correctness issues. Abstraction should be a simple process that is "obviously correct." The framework we present in this paper is applicable in the concrete such that a point-wise abstraction leads to the pushdown analyses in the literature.

This paper gives a common, simple framework to derive both CFA2 and PDCFA using a recipe to apply pushdown analysis techniques to arbitrary semantics in a manner similar to AAM. To exercise the recipe further, we give a novel analysis for delimited, composable control.

2 What to expect

CFA2 uses a technique called "summarization" from Muchnick and Jones [1981, Chapter 7], which is synonymous with the ϵ-closure graph that PDCFA constructs. Summarization algorithms need not be restricted to languages with well-bracketed calls and returns. We can adopt the technique for higher precision in the common case but still handle difficult cases such as first-class control. This was shown for the `call-with-current-continuation` (a.k.a. `call/cc`) operator in Vardoulakis and Shivers [2011a]. This impressive work illuminated the fact that we can harness the enhanced technology of pushdown analyses in non-pushdown models of computation. Doing this sacrifices call/return matching in the general case, but in practice the precision is much better than the alternative regular model that, say, AAM would provide.

A downside of the work providing `call/cc` is that it was an algorithmic change to the already complicated CFA2 — there was no recipe for how to do this for one's favorite control operator. This paper seeks to do just that with an operational view of what summarization is, in essence. In other words, we give a concrete semantics to the tricks that the analyses in the literature use in the abstract, and maintain the original meaning of the language. We also give intuitive analogies to well-established ideas/techniques so that the working semanticist can write a pushdown analysis for her language. In order to demonstrate the applicability of this viewpoint, we show a new analysis for a language with composable control. All of the semantics modeled in this paper are implemented in full detail in PLT Redex [Felleisen et al., 2009] and available online[2].

There are common underpinnings of PDCFA and CFA2 that can be embodied as concrete machinery in the programming language semantics:

1. breaking the recursive structure of continuations by indirecting them through a table with appropriately precise keys, and
2. memoization.

[2] `http://github.com/ianj/concrete-summaries`

Once the machine semantics is in this form, simple point-wise abstraction leads to the summarization algorithms that we see in the literature.

The remaining sections of the paper are

- section 3: we derive a cousin of PDCFA
- section 4: we make additions to the previous semantics to get a direct-style CFA2 without first-class control
- section 5: we give a novel analysis of delimited and composable first-class control.

3 Deriving PDCFA

PDCFA does not have some orthogonal semantic components that CFA2 features to improve precision and is thus the simpler of the two. The key to their method is to notice in languages without stack-capturing features, the continuation is only modified a frame at a time. If the state space without the stack is finite (and it can be made finite with an approximation ala AAM), then the model falls directly into the realm of pushdown systems. They thus recast the problem in terms of pushdown systems. The whole of their machinery is then computing the pushdown system on-the-fly, only considering states reachable from the root (initial) state. They call the main data-structure for this a Dyck state graph. To match nodes that push frames with later nodes following the pop of that frame, they additionally compute an ϵ-closure graph, which keeps edges between nodes that have a net-zero stack change path between them. We skip the detour to pushdown systems altogether and show an operational understanding of the resulting analysis.

We start by deriving PDCFA from an operational semantics for the untyped lambda calculus (figure 1). The semantic spaces for the machine follow.

$$\begin{aligned}
e \in \mathit{Expr} &= x \mid (e\ e) \mid \lambda x.e \\
\varsigma \in \mathit{State} &= (\mathit{Expr} \times \mathit{Env}) \times \mathit{Store} \times \mathit{Kont} \\
v \in \mathit{Value} &::= (\lambda x.e, \rho) \\
\kappa \in \mathit{Kont} &::= \mathbf{mt} \mid \mathbf{ar}(e, \rho, \kappa) \mid \mathbf{fn}(v, \kappa) \\
\rho \in \mathit{Env} &= \mathit{Var} \to \mathit{Addr} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \to \wp(\mathit{Value}) \\
x \in \mathit{Var} &\text{ an infinite set} \\
a \in \mathit{Addr} &\text{ an infinite set}
\end{aligned}$$

The *Addr* space is what controls the precision of the model. For a concrete semantics, we require the allocation meta-function $\mathit{alloc} : \mathit{State} \to \mathit{Addr}$ to return fresh addresses ($a \notin \mathrm{dom}(\sigma)$), but any choice of address is sound. If *alloc* only uses addresses from a finite subset of *Addr*, then the state space without continuations is finite, and the space of contination frames is finite (thus the pushdown system interpretation is apt). This meta-function along with the commitment to having no recursive data-structures is the key to the AAM technique. All recursion can be expressed by indirecting through addresses into a store — tying Landin's knot. We borrow this technique on top of our own.

$$
\begin{array}{r|l}
\multicolumn{2}{c}{\varsigma \longmapsto \varsigma' \text{ where } a = \mathit{alloc}(\varsigma)} \\
\hline
\langle (x, \rho), \sigma, \kappa \rangle & \langle v, \sigma, \kappa \rangle \text{ if } v \in \sigma(\rho(x)) \\
\langle ((e_0\ e_1), \rho), \sigma, \kappa \rangle & \langle (e_0, \rho), \sigma, \mathbf{ar}(e_1, \rho, \kappa) \rangle \\
\langle v, \sigma, \mathbf{ar}(e, \rho, \kappa) \rangle & \langle (e, \rho), \sigma, \mathbf{fn}(v, \kappa) \rangle \\
\langle v, \sigma, \mathbf{fn}((\lambda x.e, \rho), \kappa) \rangle & \langle (e, \rho[x \mapsto a]), \sigma \sqcup [a \mapsto \{v\}], \kappa \rangle
\end{array}
$$

Fig. 1. The CESK machine

Tracking return points: the magic of the method is in keeping a table of continuation segments for each function and store pair, where the continuation is segmented at function boundaries. This closely mirrors the technique of store-allocating continuations used in AAM. Instead of deferring to the table (store) for the remainder of a continuation for each frame, we only have indirections at function call boundaries (see rules 4 and 5 in figure 2 for saving and restoring continuations). Note that since AAM is about finitizing the state space, this step itself fits within the AAM method, since continuations within functions truncated at call boundaries are bounded by the nesting depth of those functions, thereby making the continuation space finite. In the monovariant case, the number of continuations is still linear in the size of the program.

The tail of a continuation in the context of a function will contain the stackless context in which the function was called, in order to link up with the proper call-site(s). The context includes the function (or unique label of the function), the environment, and the store. Notice that this choice can be viewed as a special allocation strategy for continuations in the AAM viewpoint, but we separate the table of continuations from the store since continuations themselves contain stores in **rt** frames — this leads to a recursive data-structure that we are trying to avoid. The store is an important ingredient to maintaining enough precision to keep a pushdown abstraction. The semantic spaces for the machine are modified thusly:

$$
\begin{aligned}
\varsigma &\in \mathit{State} = (\mathit{Expr} \times \mathit{Env}) \times \mathit{Store} \times \mathit{Kont} \times \mathit{KTable} \times \mathit{Memo} \\
\kappa &\in \mathit{Kont} ::= \mathbf{mt} \mid \mathbf{ar}(e, \rho, \kappa) \mid \mathbf{fn}(v, \kappa) \mid \mathbf{rt}((e, \rho), \sigma) \\
\Xi &\in \mathit{KTable} = (\mathit{Expr} \times \mathit{Env}) \times \mathit{Store} \to \wp(\mathit{Kont}) \\
M &\in \mathit{Memo} = (\mathit{Expr} \times \mathit{Env}) \times \mathit{Store} \to \wp(\mathit{Value})
\end{aligned}
$$

The role of summaries: notice the additional *Memo* component. A "summary edge" in CFA2 or equivalently an ϵ-edge in PDCFA is an edge from the source of a push edge to the target of the matching pop edge. “Matching” here means there is a path through machine reductions that don’t change the stack, or through summary edges. In our model, we only “push” when we call a function, and we only “pop” when we return with a value. There is an analogy to something more operational: summary edges embody memoization (see the last rule in figure 2). Instead of following an entire path through a call to return a value, we simply jump from the call to the return with the result of the call (the second case of rule 4).

$$
\begin{array}{r|l}
\multicolumn{2}{c}{\varsigma \longmapsto \varsigma' \text{ where } a = \mathit{alloc}(\varsigma)} \\
\hline
\langle (x,\rho), \sigma, \kappa, \Xi, M \rangle & \langle v, \sigma, \kappa, \Xi, M \rangle \text{ if } v \in \sigma(\rho(x)) \\
\langle ((e_0\ e_1), \rho), \sigma, \kappa, \Xi, M \rangle & \langle (e_0, \rho), \sigma, \mathbf{ar}(e_1, \rho, \kappa), \Xi, M \rangle \\
\langle v, \sigma, \mathbf{ar}(e, \rho, \kappa), \Xi, M \rangle & \langle (e, \rho), \sigma, \mathbf{fn}(v, \kappa), \Xi, M \rangle \\
\langle v, \sigma, \mathbf{fn}((\lambda x.e, \rho), \kappa), \Xi, M \rangle & \langle p, \sigma', \mathbf{rt}(p, \sigma'), \Xi \sqcup [(p, \sigma') \mapsto \{\kappa\}], M \rangle \\
 & \text{or} \\
 & \langle v_{\mathit{result}}, \sigma', \kappa, \Xi, M \rangle \text{ if } v_{\mathit{result}} \in M(p, \sigma') \\
\text{where} & p = (e, \rho[x \mapsto a]) \\
 & \sigma' = \sigma \sqcup [a \mapsto \{v\}] \\
\langle v, \sigma, \mathbf{rt}(p, \sigma'), \Xi, M \rangle & \langle v, \sigma, \kappa, \Xi, M \sqcup [(p, \sigma') \mapsto \{v\}] \rangle \text{ if } \kappa \in \Xi(p, \sigma')
\end{array}
$$

Fig. 2. The summarizing tabular stack machine

To turn this semantics into PDCFA’s algorithm for constructing a Dyck state graph, given the appropriate *alloc*, we apply a widening operator to make σ, κ, and M shared amongst all states (meaning states then become represented without these compontents), in what is then called a *System*. Since these components are shared, they are always the least upper bound of any respective component that the semantics produces (the intermediate set I) in order to stay sound. We use a meta-function *wn*, “wide to narrow”, to shift between the two representations of states.

$$\begin{aligned}
\hat{\varsigma} \in \widehat{State} &= (Expr \times Env) \times Kont \\
System &= \wp(\widehat{State} \times Store) \times \wp(\widehat{State}) \times Store \times KTable \times Memo \\
\mathcal{F} &: System \to System \\
\mathcal{F}(S, F, \sigma, \Xi, M) &= (S \cup S', F', \sigma', \Xi', M') \\
\text{where } I &= \{\varsigma' \ : \ \hat{\varsigma} \in F, wn(\hat{\varsigma}, \sigma, \Xi, M) \longmapsto \varsigma'\} \\
\sigma' &= \bigsqcup\{\sigma' \ : \ wn(\_, \sigma', \_, \_) \in I\} \\
\Xi' &= \bigsqcup\{\Xi' \ : \ wn(\_, \_, \Xi', \_) \in I\} \\
M' &= \bigsqcup\{M' \ : \ wn(\_, \_, \_, M') \in I\} \\
S' &= \{(\hat{\varsigma}', \sigma') \ : \ wn(\hat{\varsigma}', \_, \_, \_) \in I\} \\
F' &= \{\hat{\varsigma}' \ : \ (\hat{\varsigma}', \_) \in S' \setminus S\} \\
wn(\langle(e, \rho), \kappa\rangle, \sigma, \Xi, M) &= \langle(e, \rho), \sigma, \kappa, \Xi, M\rangle
\end{aligned}$$

A *System* embodies the states seen and at which store, S, the frontier set of states yet to analyze, F, the shared store for the frontier, σ, the continuation table Ξ and the memo table M. All the states in the frontier are stepped with the current store, after which the next store is the least upper bound of all the resulting stores. The next frontier contains only states resulting from stepping the previous frontier that we haven't seen at this next store. Systematic techniques for a performant implementation can be found in Johnson et al. [2012].

Example: Let's consider a monovariant allocation strategy (the address for each binding is the variable name itself) to run the following example:

```
(let* ([id (λ (x) x)]
       [y (id 0)]
       [z (id 1)])
  (≤ y z))
```

Suppose we extend our semantics to allow numbers, numeric primitives and `let`. The continuation frame for a `let` contains the identifier to bind to the resulting value, along with the body of the `let` with its environment. Call the constructor of this frame **lt**.

We should expect that a pushdown analysis would predict this evaluates to true, and there are no loops in the program. 0CFA [Shivers, 1991] claims there is a loop from the second call of `id` to the first, and thusly predicts this program evaluates to true or false. PDCFA claims there is no loop, and depending on the implementation, that the result is true or false (paper), or just true (implemented)[3]. Let's take a look at the evaluation after binding `id` ($\sigma_0 = [\mathtt{id} \mapsto \{(\lambda\mathtt{x}.\mathtt{x}, \bot)\}]$):

[3] This is because the paper steps every seen state with the current store every iteration, but the implementation only steps states that need stepping.

1. `(id 0)` steps to `x` at $\mathbf{rt}(((\mathtt{x}, [\mathtt{x} \mapsto \mathtt{x}]), \sigma_0))$ with $\sigma_1 = \sigma_0[\mathtt{x} \mapsto \{0\}]$ and (let $ctx_1 = ((\mathtt{x}, [\mathtt{x} \mapsto \mathtt{x}]), \sigma_1)$, $\kappa_1 = \mathbf{lt}(\mathtt{y}, \mathtt{(let\ ([z\ (id\ 1)])\ (\leq\ y\ z))}, [\mathtt{id} \mapsto \mathtt{id}]))$ $\Xi_1 = [ctx_0 \mapsto \{\kappa_0\}]$
2. `0` at $\mathbf{rt}(ctx_1)$ steps to 0 at κ_1 and $M_1 = [ctx_1 \mapsto \{0\}]$
3. `0` at κ_1 steps to `(let ([z (id 1)]) (≤ y z))` and $\sigma_2 = \sigma_1[\mathtt{y} \mapsto \{0\}]$
4. `(let ([z (id 1)]) (≤ y z))` steps to `(id 1)` at $\mathbf{lt}(\mathtt{z}, \mathtt{(\leq\ y\ z)}, [\mathtt{id} \mapsto \mathtt{id}, \mathtt{y} \mapsto \mathtt{y}])$ (call this κ_2).
5. `(id 1)` steps to x with $\sigma_3 = \sigma_2[\mathtt{x} \mapsto \{0, 1\}]$ and (let $ctx_3 = ((\mathtt{x}, [\mathtt{x} \mapsto \mathtt{x}]), \sigma_3))$ $\Xi_3 = \Xi_1[ctx_3 \mapsto \{\kappa_2\}]$.
6. `0` or `1` at $\mathbf{rt}(ctx_3)$ steps to `0` or `1` at κ_2 and $M_3 = M_1[ctx_3 \mapsto \{0, 1\}]$.
7. `z` gets bound to $\{0, 1\}$, and `(≤ y z)` evaluates to true.

We maintained enough context to distinguish the return points of `id` to not rebind `y` to 1. When determining control flow through the expression, we consult M to continue past function calls, so there is no confusion about a back edge from the second call to `id`.

4 Deriving CFA2

CFA2 is the first published analysis of a higher-order programming language that could properly match calls and returns. We will show that it fits well into the same presentation we gave for PDCFA. Vardoulakis and Shivers had a clear goal of harnessing the extra information a pushdown model provides to produce a high-precision analysis that works well in practice. This resulted in more than just the call/return matching of the previous section, which is why we are showing the two separately. There are two orthogonal features of the semantics:

1. stack allocation for some bindings in an additional $\xi \in \mathit{Store}$
2. strong updates on stack frames for resolved non-determinism

The first of these is an addition to the stack-less context. There is a conservative pre-analysis that checks locally whether a binding will never escape, and classifies references (labeled with a distinguishing ℓ from an arbitrary space of labels) as able to use the stack frame or not. Their criteria for a binding never escaping is that it is never referenced in a function that is not its binder. This can be extended in a language with more linguistic features; see Kranz's thesis about register-allocatable bindings in the Orbit Scheme compiler [Kranz, 1988]. A stackable reference is one that appears in the body of the binding function, by which we mean not within the body of a nested function. They use the information that a binding never escapes to not bother allocating it in the heap. This has the advantage of not changing the heap, and thus leads to less propagation. The addition of these stack frames makes the analysis exponential in theory, though in practice they have been observed to decrease running time in most cases.

The second of these is to ameliorate a problem they call "fake rebinding." That is, since bindings in the abstract represent several values, we don't want to reference a variable `x` in two different places and have it resolve to two different values. In AAM, a variable reference non-deterministically steps to all possible values associated with that variable. Here we want to say that once `x` is considered to stand for value v, then all subsequent references of `x` should be v. If they aren't, it looks as if `x` was rebound; it hasn't, and thus it is a "fake rebinding." CFA2 does not step to all values on variable reference, but instead carries all its values around in superposition until they need to be observed at, say, a function call. We give a simplified semantics that is more along the AAM style, but CFA2's approach can easily be recovered from it[4].

CFA2 uses a "local semantics" that is similar to our segmenting continuations at function boundaries, but gets stuck when it gets to a point where it would need to "return." It instead appeals to an external, imperative algorithm for summarization to sew the function calls and returns together by pattern matching on the states that the local semantics produces. What we show here is almost wholly the same in character, only embodied still in terms of a machine semantics and thus more easily reasoned about, and can be run in the concrete.

We show only the significantly modified rules of the semantics in figure 3. The other rules simply carry along the extra ξ component untouched.

$$\begin{array}{r|l}
\multicolumn{2}{c}{\varsigma \longmapsto \varsigma' \text{ where } a = \mathit{alloc}(\varsigma)} \\
\hline
\langle (x^\ell, \rho), \sigma, \xi, \kappa\rangle & \langle v, \sigma, \xi', \kappa\rangle \text{ if } (\xi', v) \in \mathcal{L}(\sigma, \xi, \rho, x, \ell) \\
\langle v, \sigma, \xi, \mathbf{fn}((\lambda x.e, \rho), \kappa)\rangle & \langle (e, \rho[x \mapsto a]), \sigma', \xi', \kappa\rangle \\
\text{where} & (\sigma', \xi') = \mathit{bind}(\sigma, \xi, a, x, v)
\end{array}$$

Fig. 3. The CESξK machine

The meta-functions the semantics uses to extend and consult the heap and stack use implicit information from the pre-analysis we described above:

$$\begin{aligned}
\mathit{bind}(\sigma, \xi, a, x, v) &= \begin{cases} (\sigma, [a \mapsto \{v\}]) & \text{if } x \text{ never escapes} \\ (\sigma \sqcup [a \mapsto \{v\}], [a \mapsto \{v\}]) & \text{otherwise} \end{cases} \\
\mathcal{L}(\sigma, \xi, \rho, x, \ell) &= \begin{cases} \{(\xi[\rho(x) \mapsto \{v\}], v) \ : \ v \in \xi(\rho(x))\} & \text{if } \ell \text{ non-escaping} \\ \{(\xi, v) \ : \ v \in \sigma(\rho(x))\} & \text{otherwise} \end{cases}
\end{aligned}$$

Sidestepping fake rebinding does not need to be restricted to stackable references, but that is what CFA2 does. Indeed, as soon as the non-determinism has been determined, we could extend the stack frame so any subsequent reference means what it meant previously in the function. Look-up would then always try the stack frame before falling back on the heap.

CFA2 also includes what they called "transitive summaries" to deal with tail calls. We insert an **rt** frame at every function call. This view contends with tail calls, but we can identify tail calls easily — any call with an **rt** as its

[4] Our Redex model implements the fake rebinding strategy CFA2 itself employs

continuation is a tail call. Tail calls are important in language implementations for space complexity reasons [Clinger, 1998], but in an analysis, these concerns are less important. The repeated popping of **rt** inserted by what otherwise were tail-calls is synonymous with CFA2's transitive summaries.

Example: if we use this semantics to run through the example of the previous section, we find that with the addition of stack frames, there is no confusion about `z`'s value either, so if the operator weren't $\leq$ but instead $<$ or $+$, we could still constant fold that away.

5 Analysis of delimited, composable control

There is contention among programming language researchers whether `call/cc` should be a language primitive, since it captures the entire stack, leading to space leaks [Kiselyov, 2012]. Alternative control operators have been proposed that delimit how much of the stack to capture, such as % (read "prompt") and capture operator $\mathcal{F}$ (read "control") [Felleisen, 1988], or `reset` (equivalent to %) and `shift` [Danvy and Filinski, 1990]. However, the stacks captured by these operators always extend the stack when invoked, rather than replace it like those captured with `call/cc`. Continuations that have this extension behavior are called "composable continuations." Stack replacement is easily modeled in a regular analysis using the AAM approach, and Vardoulakis and Shivers showed it can be done in a pushdown approach (although it breaks the pushdown model). Stack extension, however, poses a new challenge for pushdown analysis, since one application of a composable continuation means pushing an unbounded amount of frames onto the stack. Vardoulakis' and Shivers' approach does not immediately apply in this situation, since their technique drops all knowledge of the stack at a continuation's invocation site; extension, however, must preserve it.

The way we have been splitting continuations at function calls has similarities to the meta-continuation approach to modeling delimited control, given in figure 4 (adapted from [Biernacki et al., 2006]). We could view each function call as inserting a prompt, and returns as aborting to the nearest prompt. Resets then insert a second-tier prompt. The changed semantic spaces for the shift/reset semantics are as follows:

$$\begin{aligned}
\varsigma \in \mathit{State} &::= \langle p, \sigma, \kappa, C\rangle \\
p \in \mathit{Point} &::= (e, \rho) \mid v \\
v \in \mathit{Value} &::= (\lambda x.e, \rho) \mid \mathbf{comp}(\kappa) \\
C \in \mathit{MKont} &::= \mathbf{mt} \mid \kappa \circ C
\end{aligned}$$

Turning this into a table-based semantics involves making prompts a point of indirection, just like function calls. Memoization also gets a new context to consider, calling a continuation, because composable continuations act like functions. The new semantic spaces are then

$\varsigma \longmapsto \varsigma'$ where $a = alloc(\varsigma)$	
$\langle((\texttt{reset}\ e), \rho), \sigma, \kappa, C\rangle$	$\langle(e, \rho), \sigma, \mathbf{mt}, \kappa \circ C\rangle$
$\langle v, \sigma, \mathbf{mt}, \kappa \circ C\rangle$	$\langle v, \sigma, \kappa, C\rangle$
$\langle((\texttt{shift}(x)e), \rho), \sigma, \kappa, C\rangle$	$\langle(e, \rho[x \mapsto a]), \sigma \sqcup [a \mapsto \{\kappa\}], \mathbf{mt}, C\rangle$
$\langle v, \sigma, \mathbf{fn}(\kappa', \kappa), C\rangle$	$\langle v, \sigma, \kappa', \kappa \circ C\rangle$

Fig. 4. Machine semantics for shift/reset

$$
\begin{aligned}
ctx \in \mathit{Context} &::= ((e, \rho), \sigma) \mid (\mathbf{comp}(\kappa), v, \sigma) \\
\kappa \in \mathit{Kont} &= \mathbf{mt} \mid \mathbf{rt}(ctx) \mid \mathbf{ar}(e, \rho, \kappa) \mid \mathbf{fn}(v, \kappa) \\
C \in \mathit{MKont} &::= \mathbf{mt} \mid \#(ctx) \\
\Xi \in \mathit{KTable} &= \mathit{Context} \to (\wp(\mathit{Kont} \times \mathit{MKont}) \cup \mathit{Kont}) \\
M \in \mathit{Memo} &= \mathit{Context} \to \wp(\mathit{Value})
\end{aligned}
$$

Figure 5 has what one might naturally write using just this information. Unfortunately, since **rt** frames contain a store, and continuations can now appear in the store, this introduces a circularity that could cause the analysis to never terminate. For example, a loop that continually captures continuations would introduce unboundedly many continuation values due to the ever-growing store. This means the store is not finite height, and there may be no fixed point.

Possible fixes to non-termination: The simplest fix to the circularity would be to strip stores out of the context of the **rt** frame in a captured continuation, and additionally consider contexts with a stripped store an abstraction of the same context with any store. This is a fairly brutal approximation of a captured continuation, so an alternative is to make this tunable with our precision-tuning friend, *alloc*. Then, instead of entirely removing the store component of the **rt** context, we can replace it with an address of possible stores that it would approximate.

$\varsigma \longmapsto \varsigma'$ where $a = alloc(\varsigma)$	
$\langle((\texttt{reset}\ e), \rho), \sigma, \kappa, C, \Xi, M\rangle$	$\langle p, \sigma, \mathbf{mt}, \#(ctx), \Xi \sqcup [ctx \mapsto \{(\kappa, C)\}], M\rangle$
where	$p = (e, \rho),\ ctx = (p, \sigma).$
$\langle v, \sigma, \mathbf{mt}, \#(ctx), \Xi, M\rangle$	$\langle v, \sigma, \kappa, C, \Xi, M \sqcup [ctx \mapsto \{v\}]\rangle$ if $(\kappa, C) \in \Xi(ctx)$
$\langle((\texttt{shift}(x)e), \rho), \sigma, \kappa, C, \Xi, M\rangle$	$\langle(e, \rho[x \mapsto a]), \sigma \sqcup [a \mapsto \{\mathbf{comp}(\kappa)\}], \mathbf{mt}, C, \Xi, M\rangle$
$\langle v, \sigma, \mathbf{fn}(\mathbf{comp}(\kappa'), \kappa), C, \Xi, M\rangle$	$\langle v, \sigma, \kappa', \#(ctx), \Xi', M\rangle$
where	$ctx = (\kappa, v, \sigma')$
	$\Xi' = \Xi \sqcup [ctx \mapsto \{(\kappa, C)\}]$
$\langle v, \sigma, \mathbf{fn}(\mathbf{comp}(\kappa'), \kappa), C, \Xi, M\rangle$	$\langle v', \sigma, \kappa, C, \Xi, M\rangle$ if $v' \in M(\kappa', v, \sigma)$

Fig. 5. Faulty table-based semantics for shift/reset

The new indirection possibility changes *Context* to also include a where there was previously a σ (though contexts for continuation calls remain unchanged), and *KTable* now additionally maps *Addr* to a set of stores. The rules that change are presented in figure 6.

$$\begin{aligned}
ctx \in \mathit{Context} &::= ((e,\rho),\sigma) \mid ((e,\rho),a) \mid (\mathbf{comp}(\kappa),v,\sigma) \\
sctx \in \mathit{StorableContext} &::= ((e,\rho),a) \\
\hat{\kappa} \in \widehat{\mathit{Kont}} &= \mathbf{rt}(sctx) \mid \ldots \\
v \in \mathit{Value} &= (\lambda x.e,\rho) \mid \mathbf{comp}(\hat{\kappa}) \\
C \in \mathit{MKont} &::= \mathbf{mt} \mid \#(ctx) \\
\Xi \in \mathit{KTable} &= (\mathit{Context} \to (\wp(\mathit{Kont} \times \mathit{MKont}) \cup \mathit{Kont})) \\
&\quad \cup (\mathit{Addr} \to \wp(\mathit{Store}))
\end{aligned}$$

$$\begin{array}{r|l}
\multicolumn{2}{c}{\varsigma \longmapsto \varsigma' \text{ where } a = alloc(\varsigma)} \\
\hline
\langle((\mathtt{shift}(x)e),\rho),\sigma,\kappa,C,\Xi,M\rangle & \langle(e,\rho[x\mapsto a]),\sigma\sqcup[a\mapsto\{\mathbf{comp}(\kappa')\}],\mathbf{mt},C,\Xi',M\rangle \\
\text{where} & (\kappa',\Xi') = approximate(\kappa,\Xi,a) \\
\langle v,\sigma,\mathbf{rt}(ctx),C,\Xi,M\rangle & \langle v,\sigma,\kappa,C,\Xi,M'\rangle \text{ if } \kappa \in returns(\Xi,ctx) \\
\text{where} & M' = memoize(M,\Xi,ctx,v)
\end{array}$$

Fig. 6. Fixed table-based semantics for shift/reset

The meta-functions mentioned in the fixed semantics all deal with the addition of a to contexts. If the context in an **rt** frame is approximate, we must return to all the continuations known for all the contexts it approximates:

$$\begin{aligned}
returns(\Xi,((e,\rho),\sigma)) &= \Xi((e,\rho),\sigma) \\
returns(\Xi,((e,\rho),a)) &= \bigcup\{\Xi((e,\rho),\sigma) \;:\; \sigma \in \Xi(a)\}
\end{aligned}$$

At return boundaries, the memo table must add the result to all the represented contexts:=

$$\begin{aligned}
memoize(M,\Xi,((e,\rho),\sigma),v) &= M \sqcup [((e,\rho),\sigma) \mapsto \{v\}] \\
memoize(M,\Xi,((e,\rho),a),v) &= M \sqcup \bigsqcup_{\sigma\in\Xi(a)} [((e,\rho),\sigma) \mapsto \{v\}]
\end{aligned}$$

At capture time, we strip the store in the **rt** frame if there is one, and replace it with an address:

$$\begin{aligned}
\mathit{approximate}(\kappa, \Xi, a) &= (\mathit{replace}\sigma(\kappa), \mathit{add}\sigma(\kappa)) \\
\text{where } \mathit{replace}\sigma(\mathbf{mt}) &= \mathbf{mt} \\
\mathit{replace}\sigma(\mathbf{rt}(((e,\rho),\_))) &= \mathbf{rt}(((e,\rho),a)) \\
\mathit{replace}\sigma(\mathbf{ar}(e,\rho,\kappa)) &= \mathbf{ar}(e,\rho,\mathit{replace}\sigma(\kappa)) \\
\mathit{replace}\sigma(\mathbf{fn}(v,\kappa)) &= \mathbf{fn}(v,\mathit{replace}\sigma(\kappa)) \\
\mathit{add}\sigma(\mathbf{mt}) &= \Xi \\
\mathit{add}\sigma(\mathbf{rt}(((e,\rho),\sigma))) &= \Xi \sqcup [a \mapsto \{\sigma\}] \\
\mathit{add}\sigma(\mathbf{rt}(((e,\rho),a'))) &= \Xi \sqcup [a \mapsto \Xi(a')] \\
\mathit{add}\sigma(\mathbf{ar}(e,\rho,\kappa)) &= \mathit{add}\sigma(\mathbf{fn}(v,\kappa)) = \mathit{add}\sigma(\kappa)
\end{aligned}$$

The result of this viewpoint is a sound, precise, and computable semantics for a "pushdown approach" to analyzing delimited, composable control. To handle non-composable control, we can simply remove the prompts and keep *approximate*.

6 Related Work

The immediately related work is that of PDCFA [Earl et al., 2010], CFA2 [Vardoulakis and Shivers, 2011b,a], and AAM [Van Horn and Might, 2010], the first two of which we recreated in full detail. The version of CFA2 that handles `call/cc` does not handle composable control, and is dependent on a restricted CPS representation. They also had no way to tune the precision of their first class continuation approximation. Since `call/cc` can be simulated with shift/reset, this work supercedes theirs. The extended version of PDCFA that does garbage collection [Earl et al., 2012] also fits into our model, although we did not explicitly show it. The addresses that the stack keeps alive can be accumulated by "reading through" the continuation table, building up the set of addresses in each portion of the stack that we come across.

Pushdown models have existed in the first-order static analysis literature [Muchnick and Jones, 1981, Chapter 7][Reps et al., 1995], and the first-order model checking literature [Bouajjani et al., 1997], for some time. The higher-order setting imposes additional challenges that make their methods difficult to adapt. The most important constraint is that we can't know all call-sites of a function/method before the analysis begins, which their methods heavily rely on.

A new temporal logic that itself understands well-bracketing of call and returns for stating and validating propositions, NWTL [Alur et al., 2007], applies to visibly pushdown systems, which can precisely model programming languages that only have well-bracketed call and returns. The propositional form of the logic has decidable satisfiability checking, and control-flow queries are shallow propositions to pose in the logic. However, the satisfiability problem is 2Exptime-hard and the algorithm is not given in the on-the-fly form that a higher-order language would need to effectively answer control-flow queries without already knowing the answers. The logic itself is an exciting new frontier for expressing program correctness properties, and a higher-order version would be a welcome addition to the functional programmer's tool belt.

The trend of deriving static analyses from abstract machines does not stop at flow analyses. The model-checking community showed how to check temporal logic queries for collapsible pushdown automata (CPDA), or equivalently, higher-order recursion schemes, by deriving the checking algorithm from the Krivine machine [Salvati and Walukiewicz, 2011]. The expressiveness of CPDAs outweighs that of PDAs, but it is unclear how to adapt higher-order recursion schemes to model arbitrary programming language features. The method is strongly tied to the simply-typed call-by-name lambda calculus and depends on finite sized base-types.

7 Conclusion and future work

As the programming world continues to embrace behavioral values, it becomes more important to import the powerful techniques pioneered by the first-order analysis and model checking communities. CFA2 and PDCFA paved the way, and in large part inspired this work. It is our view that systematic approaches to applying the techniques are pivotal to scaling them to "real languages." We believe that the recipe that this paper set forth is a step in that direction. That is, make continuation tables keyed with enough context, and memoize at the introduced indirection points. The result in a language with well-bracketed control is a "pushdown analysis" using summarization. In a language without well-bracketed control, we are not chained to a pushdown automaton as the target of the approximation, so the techniques still apply and give better precision than regular methods.

Our goal in the future is to show that this technique is even more widely applicable than shift and reset. We conjecture that the same recipe will apply to the most intricate control operators in production languages such as in Flatt et al. [2007]. The control structures there are difficult to model even with AAM's techniques due to the ability to capture and compose continuations with arbitrarily many prompts, but once we can tackle AAM, there should be a straightforward route to extending it with the pushdown techniques of this paper.

There is also the question of clients of these analyses. Most obviously we would want to know where we can implement first-class control more efficiently. In particular, we would want to have an escape analysis to find first-class continuations that don't need to be heap-allocated, and single target abort operations that can be turned into long jumps after a computed stack unraveling. On top of optimizations, there are security analyses. Greater control of the stack gives us the ability to drill deep into context-sensitive security properties and make precise predictions.

Bibliography

R. Alur, M. Arenas, P. Barcelo, K. Etessami, N. Immerman, and L. Libkin. First-Order and temporal logics for nested words. pages 151–160, 2007.

Dariusz Biernacki, Olivier Danvy, and Chung chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006.

Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. pages 135–150. 1997.

William D. Clinger. Proper tail recursion and space efficiency. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, 1998.

Olivier Danvy and Andrzej Filinski. Abstracting control. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM, 1990.

Christopher Earl, Matthew Might, and David V. Horn. Pushdown Control-Flow analysis of Higher-Order programs. In *Proceedings of the 2010 Workshop on Scheme and Functional Programming (Scheme 2010)*, 2010.

Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 177–188. ACM, 2012.

Matthias Felleisen, Robert B. Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009.

Mattias Felleisen. The theory and practice of first-class prompts. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 180–190. ACM, 1988.

Matthew Flatt, Gang Yu, Robert B. Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 165–176. ACM, 2007.

J. Ian Johnson, Matthew Might, and David Van Horn. Optimizing abstract abstract machines. *arXiv*, abs/1211.3722, 2012.

Oleg Kiselyov. An argument against call/cc, 2012. `http://okmij.org/ftp/continuations/against-callcc.html`.

David Kranz. *ORBIT: An Optimizing Compiler For Scheme*. PhD thesis, Yale University, 1988.

Steven S. Muchnick and Neil D. Jones. *Program Flow Analysis: Theory and Applications (Prentice-Hall Software Series)*. Prentice Hall, 1981. ISBN 0-13-729681-9.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 49–61. ACM, 1995.

S. Salvati and I. Walukiewicz. Krivine machines and higher-order schemes. In *Proceedings of the 38th international conference on Automata, languages and programming - Volume Part II*, ICALP'11, pages 162–173. Springer-Verlag, 2011.

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

David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 51–62. ACM, 2010.

Dimitrios Vardoulakis and Olin Shivers. Pushdown flow analysis of first-class control. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 69–80. ACM, 2011a.

Dimitrios Vardoulakis and Olin Shivers. CFA2: a Context-Free Approach to Control-Flow Analysis. *Logical Methods in Computer Science*, 7(2:3):1–39, 2011b.