Concrete Semantics for Pushdown Analysis: The Essence of Summarization

J. Ian Johnson and David Van Horn

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

1 What to expect

This paper lays out a common framework to talk about pushdown analysis for higher-order languages. In particular, we extract the "essence" of the existing techniques into a quality of the machine semantics that specify a language's meaning, much in the same way as **?** (a method we will refer to as AAM: abstracting abstract machines). Once the machine semantics is in this form, simple pointwise abstraction leads to the summarization algorithms that we see in the literature. In effect, we give a *concrete* semantics to pushdown analysis.

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 **?**. 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[1] model that, say, AAM would provide. A downside of the work 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. Instead of deriving a "pushdown" analysis for a language with `call/cc`, we will show a new analysis for a language with the all the control operators in **?** (call it the PLT machine) in order to demonstrate the applicability of this viewpoint even in the context of complex control operators.

The remaining sections of the paper are

- section **??**: we derive a cousin of PDCFA
- section **??**: we make additions to the previous semantics to get a direct-style CFA2 without first-class control
- section **??**: we give a novel analysis of the PLT machine, a core calculus of Racket's control operations.

2 Deriving PDCFA

PDCFA is a simpler analysis than CFA2, and thus enjoys a polynomial complexity in the monovariant case. Its final $\mathcal{O}(n^6)$ form unfortunately is not a complete

[1] Regular as in regular language.

abstraction of an unbounded stack model. In this section, we show how to derive a $\mathcal{O}(n^9)$ complete abstraction[2]. These bounds are absolute worst case that would never ever happen in practice. The complete abstraction restricts value flow to fewer spurious paths, often leading to a faster analysis than a typical 0CFA.

The magic of the method is in keeping a table of (local) continuations for each function $\times$ store pair. 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. 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 callsite(s). The context includes the function (or unique label of the function), the environment, and the store. If doing a flow analysis that takes advantage of the pushdown model, the context would also have a lattice element.

Once we finitize the address space of the store, this computes a characteristic finite state machine that has only reachable states of the pushdown system the unbounded stack model embodies.

A "summary edge" that is in the literature, in terms of pushdown systems, 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 edges that don't change the stack, or through summary edges. There is an analogy to something more operational: summary edges embody memoization. Instead of following an entire path through a call to return a value, we simply jump from the call to the return (the result of the call). We are very conservative with how much of the context is necessary to memoize, so we use the entire heap. This conservativeness isn't useful for skipping any work in computing the flow analysis (so much so that we don't even bother consulting the memo table in the semantics), but it is useful for interpreting the results to talk about the behavior of the body of a function without detours to other functions. We will return to this analogy and what it means in the presence of first-class control in section **??**.

To turn this semantics into PDCFA's algorithm for constructing a Dyck state graph, we apply a widening operator $\mathcal{F}$ to make σ, κ, and M shared amongst all states. Further techniques for a performant implementation can be found in **?**.

[2] Interestingly, the available *implementations* of PDCFA incorporate the fix and are themselves $\mathcal{O}(n^9)$.

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

Fig. 1. The CESK semantic spaces

$$
\varsigma \longmapsto \varsigma' \text{ where } a = \mathit{alloc}(\varsigma)
$$

$$
\begin{array}{r|l}
\langle (x, \rho), \sigma, \kappa \rangle & \langle v, \sigma, \kappa \rangle \text{ if } v \in \sigma(\rho(x)) \\
\langle ((e_0\ e_1), \rho), \sigma, \kappa \rangle & \langle (e_0, \rho), \sigma, \mathbf{ar}(e, \rho, \kappa) \rangle \\
\langle v, \sigma, \mathbf{ar}(e, \rho, \kappa) \rangle & \langle (e, \rho), \sigma, \mathbf{fn}(v, \kappa) \rangle \\
\langle v, \sigma, \mathbf{fn}(\lambda x.e, \kappa) \rangle & \langle (e, \rho[x \mapsto a]), \sigma \sqcup [a \mapsto \{v\}], \kappa \rangle
\end{array}
$$

Fig. 2. The CESK machine

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

Fig. 3. Extended semantic spaces

$$
\varsigma \longmapsto \varsigma' \text{ where } a = \mathit{alloc}(\varsigma)
$$

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

Fig. 4. The summarizing tabular stack machine

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

3 Deriving CFA2

CFA2 is the first published analysis of a higher-order programming language that could properly match calls and returns. 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 extra features of the semantics:

1. stack allocation for some bindings
2. strong updates on stack frames for resolved nondeterminism

The first of these is an addition to the stackless context. There is a conservative pre-analysis that checks locally whether a binding will never escape, and classifies references 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 (**?**). A stackable reference is one that appears in the body of the binding function, by which we mean not within the body of 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 proved 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.

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

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

Fig. 5. The CESξK machine

$$\mathit{bind}(\sigma, \xi, a, x, v) = \begin{cases} (\sigma, [a \mapsto \{v\}]) & \text{if } x \text{ never escapes} \\ (\sigma \sqcup [a \mapsto \{v\}], [a \mapsto \{v\}]) & \text{otherwise} \end{cases}$$

$$\mathcal{L}(\sigma, \xi, \rho, x, \ell) = \begin{cases} \{(\xi[\rho(x) \mapsto \{v\}], v) \ : \ v \in \xi(\rho(x))\} & \text{if } \ell \text{ non-escaping} \\ \{(\xi, v) \ : \ v \in \sigma(\rho(x))\} & \text{otherwise} \end{cases}$$

4 Analysis of the PLT machine

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 (**?**). Alternative control operators have been proposed that delimit how much of the stack to capture, such as Felleisen's % (read "prompt") and capture operator $\mathcal{F}$ (read "control") (**?**), or Danvy and Filinski's `reset` (equivalent to %) and `shift` (**?**). 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 simple to model 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.

Composable and non-composable continuations both have their uses, as it has been shown the two are not co-expressive when it comes to space complexity **?**.

There is still merit in delimiting the stack in either case, as is argued by **?**. Their paper gives a semantics for the coexistence of composable and non-composable continuations in the presence of first-class prompts, `dynamic-wind` (protects entry/exit of a continuation with pre- and post-processors), and continuation marks. We will show how to apply pushdown analysis techniques in the presence of all this complexity.

The way we have been splitting continuations at function calls has similarities to the meta-continuation approach to modeling delimited control (**?**). We could view each function call as inserting a prompt, and returns as aborting to the nearest prompt. This view contends with tail calls, but we can identify tail calls easily - any call with a prompt as its continuation is a tail call. Tail calls are important in language implementations for space complexity reasons (**?**), but in an analysis, these concerns are less important. The repeated popping of prompts inserted by what otherwise were tail-calls is synonymous with CFA2's "transitive summaries."

We first give a CESK semantics of the PLT machine with the allocation procedure as a parameter in order to first discuss the complexities any computable abstract semantics will have. We then present the key ideas to making it a table-based semantics, and how we can abstract it to a computable approximation.

4.1 PLT's CESK semantics

The treatment of prompts as first-class values and `dynamic-wind`'s interaction with non-composable continuations requires equality tests of dynamically generated values. Because of this, once we relax the freshness condition for *alloc*, we need a semantics that can handle the possibility that two values aren't just equal or disequal - they *may* be equal. This complication is separate from the pushdown approach we are presenting, so we give it a separate presentation in this section to subsequently amend with pushdown techniques.

...TODO...

4.2 Tabular PLT semantics

Since continuations can now appear in the store, and the store is in the **rt**() continuation, we introduced a circularity in the data represented in our machine - this is precisely what the AAM approach avoids in order to have a finite model. There is currently no known model of computation that allows stack capture and replacement/composition that has decidable reachability, so we choose to approximate in this case. We represent only a bounded amount of a continuation, after which we say that any continuation matching that prefix is acceptable. The amount to represent is a tunable parameter that we simply set to be one of the local continuations we already have in our model.

5 Related Work

AAM, PDCFA, CFA2, CFA2+, Reps

6 Conclusion