Abstract

The strength of a dynamic language is also its weakness: run-time flexibility comes at the cost of compile-time predictability. Many of the hallmarks of dynamic languages such as closures, continuations, various forms of reflection, and a lack of static types make many programmers rejoice, while compiler writers, tool developers, and verification engineers lament. The dynamism of these features simply confounds statically reasoning about programs that use them. Consequently, static analyses for dynamic languages are far, far between, and seldom sound.

The “abstracting abstract machines” (AAM) approach to constructing static analyses has recently been proposed as a method to ameliorate the difficulty of designing analyses for such language features. The approach, so called because it derives a function for the sound and computable approximation of program behavior starting from the abstract machine semantics of a language, provides a viable approach to dynamic language analysis since all that is required is a machine description of the interpreter.

The AAM recipe as originally described produces finite state abstractions: the behavior of a program is approximated as a finite state machine. Such a model is inherently imprecise when it comes to reasoning about the control stack of the interpreter: a finite state machine cannot faithfully represent a stack. Recent advances have shown that higher-order programs can be approximated with pushdown systems. However, such models, founded in automata theory, either breakdown or require significant engineering in the face of dynamic language features that inspect or modify the run-time stack, which previously have required either breakdown or require significant engineering in the face of dynamic language features that inspect or modify the run-time stack, which previously have required significant technical innovations.

In this paper, we tackle the problem of bringing pushdown flow analysis to the domain of dynamic language features. We revise the abstracting abstract machines technique to target the stronger computational model of pushdown systems. In place of automata theory, we use only abstract machines and memoization. As case studies, we show the technique applies to a language with closures, garbage collection, stack-inspection, and first-class composable continuations.

1. Introduction

Good static analyses use a combination of abstraction techniques, economical data structures, and a lot of engineering. The cited exemplary works stand out from a vast amount of work attacking the problem of statically analyzing languages like C. Dynamic languages do not yet have such gems. The problem space is different, bigger, and full of new challenges. The traditional technique of pushing abstract values around a graph to get an analysis will not work. The first problem we must solve is, “what graph?” as control-flow is now part of the problem domain. Second, features like stack inspection and first-class continuations are not easily shoe-horned into a CFG representation of a program’s behavior. We need a new approach.

Luckily, there is an alternative to the CFG approach to analysis construction that is based instead on abstract machines, which are one step away from interpreters (they are interderivable in several instances). This alternative, called abstracting abstract machines (AAM), is a simple idea that is generally applicable to even the most dynamic of languages, e.g., JavaScript. A downside is that all effective instantiations of AAM are finite state approximations. Finite state techniques cannot precisely predict where a method or function call will return. Dynamic languages have more sources for imprecision than non-dynamic languages (e.g., reflection, computed fields, runtime linking, eval) that all need proper treatment in the abstract. If we can’t have precision in the presence of statically unknowable behavior, we should at least be able to contain it in the states it actually affects. Imprecise control flow due to finite state abstractions is an unacceptable containment mechanism. It opens the flood gate to imprecision flowing everywhere through analyses’ predictions. It is also a solvable problem.

We extend the AAM technique to computably handle infinite state spaces by adapting pushdown abstraction methods to abstract machines. The unbounded stack of pushdown systems is the mechanism to precisely match calls and returns. We demonstrate the essence of our pushdown analysis construction by first applying the AAM technique to a call-by-value functional language and then revising the derivation to incorporate an exact representation of the control stack. We then show how the approach scales to stack-reflecting language features such as garbage collection and stack inspection, and stack-reifying features in the form of first-class delimited control operators. These case studies show that the approach is robust in the presence of features that need to inspect or alter the run-time stack, which previously have required significant technical innovations.

Our approach appeals to operational intuitions rather than automata theory to justify the approach. The intention is that the only prerequisite to designing a pushdown analysis for a dynamic language is some experience implementing interpreters; expertise in automata theory or abstract domains is unneeded. With simple but powerful tools in a large audience’s toolbox, we hope this work will help the community work towards technical feats like Astrée. The large problem domain of analyzing dynamic languages needs an army to tackle it, and we believe that AAM can feed that army.

2. The AAM methodology

Abstract machines are a versatile, clear and concise way to describe the semantics of programming languages. They hit a sweet spot in terms of level of discourse, and viable implementation strategy. First year graduate students learn programming language semantics with abstract machines. They also turn out to be fairly simple to repurpose into static analyses for the languages they implement, via the Abstracting Abstract Machines methodology. The basic idea is that abstract machines implement a language’s concrete semantics, so we transform them slightly so that they also implement a language’s abstract semantics (thus “abstracting” abstract machines).
AAM is founded on three ideas:

1. concrete and abstract semantics ideally should use the same code, for correctness and testing purposes,

2. the level of abstraction should be a tunable parameter,

3. both of the above are achievable with a slight change to the abstract machine’s state representation.

The first two points are the philosophy of AAM: correctness through simplicity, reusability, and sanity checking with concrete semantics. The final point is the machinery that we recount in this section.

2.1 The CESK\textsuperscript{*} machine schema

The case studies in this paper have full implementations in PLT redex \cite{Plt-redex} available online\footnote{http://github.com/dvanhorn/aac}. They all build off the call-by-value untyped lambda calculus, whose semantics we recall in small-step reduction semantics style:

\[
E((\lambda x. e) v) \longrightarrow_{β} E[[v/x] e]
\]

where \( E \in \text{EvalContext} := [] \mid (E e) \mid (v E) \)
\( e \in \text{Expr} := \varepsilon \mid (e e) \mid v \)
\( v \in \text{Value} := \lambda x.e \)

Although concise and well-structured for mathematical proofs, this semantics does not reflect what an effective implementation looks like. The context decomposition to find a redex (left hand side of \( \longrightarrow_{β} \)), substitution and re-plugging are better understood as a process that a machine goes through to perform a step. We thus start with an abstract machine that manages these steps with a special state representation: the CESK machine\footnote{CESK stands for control string (or code), environment, store, and continuation (but with a \( k \) to distinguish from the first \( c \)). The continuation is sometimes called the stack.}. The environment and store work together to represent ongoing substitutions, and the continuation represents when evaluation is with respect to decomposing and re-plugging a context. The state space is in Figure 1. The standard CESK machine has a store that maps freshly allocated addresses to single values. An address

\[
\text{CESK} \rightarrow \text{Addr}
\]

Allocation is key for both precision and abstraction power. Fresh allocation gives us the full lambda calculus – everything is undecidable. If the allocator has a finite codomain, and there are no recursive datatypes in a state’s representation, then the state space is finite – the machine is a finite state machine. If the allocator freshly allocates addresses for the stack representation, but is finite for everything else, the semantics is indistinguishable from just using the recursive representation of the stack. If we have just the stack as an unbounded data structure, then the machine has finitely many stack frames and finitely many other components of the state – the machine is a pushdown system.

AAM slightly modifies the store to map arbitrarily allocated addresses (reuse allowed) to a set of values. That way, if an address is reused then the previous tenant of the address is not forgotten, just merged as a possible result when the address is dereferenced. The allocation function is a parameter; the “machine” AAM produces is really a machine schema, since we can vary the allocator to produce both and anything in between a concrete and abstract semantics. That said, we will still refer to the refactored semantics as machines going forward. The semantics of this CESK machine is shown in Figure 2. The only differences from the standard

\[
\text{ CESK } := (e, ρ, σ, k) \quad \text{ Value } := (f, p)
\]

The runaway recursion is in Kont: the tail of the continuation consists recursive. So then conses of frames in Kont instead allocate an address, update the heap with the recursive value, and use the address in place of the recursive value. To help an alloc : State → Addr function choose its addresses, the state space can be extended with an arbitrary pointed space that can be updated each step. The original AAM paper calls this pointed space and update function \( Time, t_0 \) and tick respectively. The point, \( t_0 \), is for the initial state’s timestamp. Later work on widening \cite{15} suggests less arbitrary constructions, and to think of \( Time \) as a space of abstract traces, though the “abstraction” need not be sound \cite{23}. The timestamp machinery is important to implementing specific allocation schemes, but is primarily noise for the exposition in this paper, so we omit it past the first couple demonstrations.

The new semantic spaces in Figure 3 form the CESK\textsuperscript{*} machine. The semantics of this machine follow the weak update and non-deterministic lookup principles of AAM in Figure 4.

If we run the CESK\textsuperscript{*} semantics to explore all possible states, we get a sound approximation of all paths that the CESK machine will explore. This paper will give a more focused view of the Kont component. We said that when just Kont is unbounded, we have a pushdown system. Pushdown systems cannot be naively

\[
\text{Figure 1. CESK semantic spaces}
\]

\[
\text{Figure 2. CESK machine}
\]

\[
\text{Figure 3. CESK\textsuperscript{*} semantic spaces}
\]

\[
\text{Removing recursion: } \text{ In order to make } \longrightarrow \text{ finite and therefore have a computable graph, we must finitize the recursive spaces in which the machine creates new values. AAM dictates that finitization can be centralized to one place, address allocation, by redirecting values in recursive positions through the store. Expressions, though recursive, do not need this step because they are only destructed. The Expr space is finite for each program, the size of which is the number of subexpressions in the program. The runaway recursion is in Kont: the tail of the continuation consists recursive. So then conses of frames in Kont instead allocate an address, update the heap with the recursive value, and use the address in place of the recursive value. To help an alloc : State → Addr function choose its addresses, the state space can be extended with an arbitrary pointed space that can be updated each step. The original AAM paper calls this pointed space and update function (Time, t_0) and tick respectively. The point, t_0, is for the initial state’s timestamp. Later work on widening \cite{15} suggests less arbitrary constructions, and to think of Time as a space of abstract traces, though the “abstraction” need not be sound \cite{23}. The timestamp machinery is important to implementing specific allocation schemes, but is primarily noise for the exposition in this paper, so we omit it past the first couple demonstrations. The new semantic spaces in Figure 3 form the CESK\textsuperscript{*} machine. The semantics of this machine follow the weak update and non-deterministic lookup principles of AAM in Figure 4.}

\[
\text{If we run the CESK\textsuperscript{*} semantics to explore all possible states, we get a sound approximation of all paths that the CESK machine will explore. This paper will give a more focused view of the Kont component. We said that when just Kont is unbounded, we have a pushdown system. Pushdown systems cannot be naively}
\]
run} to find all states and describe all paths the CESK machine can explore; the state space is infinite, therefore this strategy may not terminate. The pushdown limitation is special because we can always recognize non-termination, stop, and describe the entire state space. We show that a simple change in state representation can provide this functionality. We regain the ability to just run the semantics and get a finite object that describes all possible paths in the CESK machine, but with better precision than before.

3. A refinement for exact stacks

We can exactly represent the stack in the CESK^*_t machine with a modified allocation scheme for stacks. The key idea is that if the address is “precise enough,” then every path that leads to the allocation will proceed exactly the same way until the address is dereferenced.

“Precise enough”: For the CESK^*_t machine, every function evaluation evaluates the same way, regardless of the stack. We should then represent the stack addresses as the components of a function call. The one place in the CESK^*_t machine that continuations are allocated is at (e_0 e_1) evaluation. The expression itself, the environment, the store and the timestamp are necessary components for evaluating (e_0 e_1), so then we just represent the stack address as those four things. The stack is not relevant for its evaluation, so we do not want to store the stack addresses in the same store – that would also lead to a recursive heap structure. We will call this new table \( \Xi \), because it looks like a stack.

By not storing the continuations in the value store, we separate “relevant” components from “irrelevant” components. We split the stack store from the value store and use only the value store in stack addresses. Stack addresses generally describe the relevant context that lead to their allocation, so we will refer to them henceforth as contexts. The resulting state space is updated here:

\[
\text{State} = \text{CESK}_t \times \text{KStore} \\
\text{Storeable} = \text{Value} \\
\kappa \in \text{Kont} ::= \epsilon \mid \phi;\tau \\
\tau \in \text{Context} ::= (e, \rho, \sigma)_t \\
\Xi \in \text{KStore} = \text{Context} \rightarrow \rho(\text{Kont})
\]

The semantics is modified slightly in Figure 5 to use \( \Xi \) instead of \( \sigma \) for continuation allocation and lookup. Given finite allocation, contexts are drawn from a finite space, but are still precise enough to describe an unbounded stack: they hold all the relevant components to find which stacks are possible. The computed \( \text{------} \) relation thus represents the full description of a pushdown system of reachable states (and the set of paths). Of course this semantics does not always define a pushdown system since \( \text{alloc} \) can have an unbounded codomain. The correctness claim is therefore a correspondence between the same machine but with an unbounded stack, no \( \Xi \), and \( \text{alloc}, \text{tick} \) functions that behave the same disregarding

\[
\xi \mapsto \xi' \quad a = \text{alloc}(\xi) \quad u = \text{tick}(\xi)
\]

\[
(x, \rho, \sigma, \kappa)_t, (v, \rho, \sigma, \kappa)_t, (v, \rho, \sigma, \kappa)_t, (v, \rho, \sigma, \kappa)_t
\]

\[
\xi, \Xi \mapsto \xi', \Xi' \quad a = \text{alloc}(\xi, \Xi) \quad u = \text{tick}(\xi, \Xi)
\]

\[
(x, \rho, \sigma, \kappa)_t, \Xi, (v, \rho, \sigma, \kappa)_t, \Xi, (v, \rho, \sigma, \kappa)_t, \Xi, (v, \rho, \sigma, \kappa)_t
\]

\[
\xi \in \text{CESK} = \{v, \rho, \sigma, \iota, \kappa\} \\
\iota \in L\text{Kont} = \text{Frame}^* \\
\kappa \in \text{Kont} ::= \epsilon \mid \tau
\]

3.1 Engineered semantics for efficiency

We cover three optimizations that may be employed to accelerate the fixed-point computation.

1. Observe that \( \Xi \) can be made global with no loss in precision; it will not need to be stored in the frontier or set of seen states.
2. Continuations can be “chunked” more coarsely at function boundaries instead of at each frame in order to minimize table lookups.
3. Since evaluation is the same regardless of the stack, we can memoize results to short-circuit to the answer.

This last optimization will be covered in more detail in section 6. From here on, this paper will not explicitly mention timestamps.

A secondary motivation for the representation change is that flow analyses commonly split control-flow graphs at function call boundaries to enable the combination of intra- and inter-procedural analyses. In an abstract machine, this split looks like installing a continuation prompt at function calls. We borrow a representation from literature on delimited continuations to split the continuation into two components: the continuation and meta-continuation. Our delimiters are special since each continuation “chunk” until the next prompt has bounded length. The bound is roughly the deepest nesting depth of an expression in functions’ bodies. Instead of “continuation” and “meta-continuation” then, we will use terminology from CFA2 and call the top chunk a “local continuation,” and the rest the “continuation.”

The resulting shuffling of the semantics to accommodate this new representation is in Figure 7. The extension to \( \Xi \) happens in a different rule – function entry – so the shape of the context changes to hold the function, argument, and store. We have a choice of whether to introduce an administrative step to dereference \( \Xi \) once \( \iota \) is empty, or to use a helper metafunction to describe a “pop” of both \( \iota \) and \( \kappa \). Suppose we choose the second because the resulting semantics has a 1-to-1 correspondence with the previous semantics.

A first attempt might land us here:

\[
\text{pop}(\phi;\kappa, \Xi) = \{ (\phi, \iota, \kappa) \} \\
\text{pop}(\epsilon, \tau, \Xi) = \{ (\phi, \iota, \kappa) : \phi;\kappa \in \Xi(\tau) \}
\]

\text{Since the continuation is either } \epsilon \text{ or a context, CFA2 calls these “entries” to mean execution entry into the program } (\epsilon) \text{ or a function } (\tau). \text{ One can also understand these as entries in a table } (\Xi). \text{ We stay with the “continuation” nomenclature because they represent full continuations.}
CFA2 employs the first strategy and calls it “transitive summaries.”

However, tail calls make the dereferenced pop contexts it dereferences in order to guard against divergence. So it’s possible that abstraction makes the store grow monotonically in a finite space, will diverge chasing these contexts. Now we chosen the first strategy, the issue of divergence is delegated to that recur down the stack. Recursive metafunctions have a semantically as well, hence fair game for AAM. And, they should always terminate (otherwise the semantics is hosed). We can think of a simple pattern-matching recursive function as a set of rewrite rules that apply repeatedly until it reaches a result. Interpreted via AAM, non-deterministic metafunction evaluation leads to a set of possible results.

The finite restriction on the state space carries over to metafunction inputs, so we can always detect infinite loops that abstraction may have introduced and bail out of that execution path. Specifically, a metafunction call can be seen as an initial state, s, that will evaluate through the metafunction’s rewrite rules ⇔ to compute all terminal states (outputs):

\[ terminal: \forall A. \text{relation } A \times A \rightarrow \varphi(A) \]

\[ terminal(\cdot \Rightarrow, s) = terminal^*(\varnothing, \langle s \rangle, \varnothing) \]

where \( terminal^*(S, \varnothing, T) = T \)

\[ terminal^*(S, F, T) = terminal^*(S \cup F, F', T \cup T') \]

where \( T = \bigcup_{s \in F} post(s) \# \varnothing \rightarrow \{ s \} \)

This definition is a typical worklist algorithm. It builds the set of terminal terms, T, by exploring the frontier (or worklist), F, and only adding terms to the frontier that have not been seen, as represented by S. If S has no more steps, \( post(s) \) will be empty, meaning s should be added to the terminal set T. Note that it is possible for metafunctions’ rewrite rules to themselves use metafunctions, but the seen set (S) for terminal must be dynamically bounded – it cannot restart at \( \varnothing \) upon reentry. Without this precaution, the host language will exceed its stack limits when an infinite path is explored, rather than bail out.

### 4.1 Case study for stack traversal: GC

Garbage collection is an example of a language feature that needs to crawl the stack, specifically to find live addresses. We are interested in garbage collection because it can give massive precision boosts to analyses \([11, 24]\). The following function will produce the set of live addresses in the stack:

\[ KLLC : \text{Frame}^* \rightarrow \varphi(\text{Addr}) \]

\[ KLLC^*(\hat{\epsilon}, L) = L \]

\[ KLLC^*(\hat{\phi}, \hat{L}, L) = KLLC^*(\hat{\epsilon}, L \cup T(\varnothing)) \]

where \( T(\text{appL}(\epsilon, \rho)) = T(\text{appR}(\epsilon, \rho)) = T(\epsilon, \rho) \)

When interpreted via AAM, the continuation is indirected through \( \Xi \) and leads to multiple results, and possibly loops through \( \Xi \). Thus this is more properly understood as

\[ KLLC^*(\hat{\epsilon}, \hat{\epsilon}, L) = L \]

\[ KLLC^*(\hat{\epsilon}, L, L) \rightarrow \rightarrow \]

\[ KLLC^*(\hat{\epsilon}, L) \rightarrow \rightarrow \]

\[ KLLC^*(\hat{\epsilon}, L \cup T(\varnothing)) \]

A garbage collecting semantics can choose to collect the heap with respect to each live set (call this \( \Gamma^* \)), or, soundly, collect with respect to their union (call this \( \Gamma^* \)). On the other hand we could have

4CFA2 employs the first strategy and calls it “transitive summaries.”
tighter collections but more possible states, and on the other hand we can leave some precision behind in the hope that the state space will be smaller. In the general idea of relevance versus irrelevance, the continuation’s live addresses are relevant to execution, but are already implicitly represented in contexts because they must be mapped in the store’s domain.

A state is “collected” only if live addresses remain in the domain of \( \sigma \). We say a value \( v \in \sigma(a) \) is live if \( a \) is live. If a value is live, any addresses it touches are live; this is captured by the computation in \( \mathcal{R} \):

\[
\mathcal{R}(\text{root, } \sigma) = \{ b : a \in \text{root, } a \rightarrow \sigma b \} \\
v \in \sigma(a) \quad b \in T(v) \quad a \sim_\sigma b
\]

So the two collection methods are as follows. Exact GC produces different collected states based on the possible stacks’ live addresses:

\[
\Gamma^*(\zeta, \Xi) \equiv \{ \zeta : \zeta \sigma \in \text{live}^*(\zeta, \Xi) \}
\]

\[
\text{live}^*\{ (e, \rho, \sigma, k), \Xi \equiv \{ \mathcal{R}(T(e, \rho) \cup L, \sigma) : L \in K \mathcal{L}(\Xi, k) \}
\]

\[
\zeta, \Xi \mapsto \zeta', \Xi' \quad \zeta' \in \Gamma^*(\zeta', \Xi')
\]

And inexact GC produces a single state that collects based on all (known) stacks’ live addresses:

\[
\Gamma(\zeta, \Xi) \equiv \{ \zeta : \zeta \sigma \in \text{live}(\zeta, \Xi) \}
\]

\[
\text{live}(e, \rho, \sigma, k, \Xi) \equiv \mathcal{R}(T(e, \rho) \cup \{ K \mathcal{L}(\Xi, k), \sigma \})
\]

\[
\zeta, \Xi \mapsto \zeta', \Xi' \quad \zeta' \in \Gamma(\zeta', \Xi')
\]

And the set subtraction is to say that granted permissions do not need to be checked farther down the stack.

Continuation marks respect tail calls and have an interface that abstracts over the stack implementation. Each stack frame added to the continuation carries the permission map. The empty continuation also carries a permission map. Crucially, the added constructs do not add frames to the stack; instead, they update the permission map in the top frame, or if empty, the empty continuation’s permission map.

\[
m \in \text{PermissionMap} = \text{Permissions} \rightarrow GD
\]

\[
gd \in GD \equiv \text{Grant} | \text{Deny}
\]

\[
\kappa \in \text{Kont} \equiv \epsilon^m | \phi^m \kappa
\]

Update for continuation marks:

\[
m[P \rightarrow gd] = \lambda x. x \in P \rightarrow gd, m(x)
\]

\[
m[P \rightarrow gd] = \lambda x. x \in P \rightarrow m(x), gd
\]

The abstract version of the semantics has one change on top of the usual continuation store. The test rules are now

\[
\text{test } P e_0 e_1 \rho \sigma \kappa, \Xi \equiv \{ e_0, \rho, \sigma, \kappa, \Xi \} \text{ if } True \in \overline{\mathcal{O}_K}(\Xi, P, k) \equiv \{ e_1, \rho, \sigma, \kappa, \Xi \} \text{ if } False \in \overline{\mathcal{O}_K}(\Xi, P, k)
\]

where the a new \( \overline{\mathcal{O}_K} \) function uses terminal and rewrite rules:

\[
\overline{\mathcal{O}_K}^* (\Xi, \rho, \kappa) \equiv \text{terminal} (\rightarrow, \overline{\mathcal{O}_K}^*(\Xi, P, k))
\]

\[
\overline{\mathcal{O}_K}^* (\Xi, P, \epsilon^m) \equiv \text{pass}(P, m)
\]

\[
\overline{\mathcal{O}_K}^* (\Xi, P, \phi^\tau) \equiv b \text{ where } b \in \{ \text{pass}(P, m) \text{ and } b : \kappa \in \Xi(\tau), \}
\]

This definition works fine with the reentrant terminal function with a dynamically bound seen set, but otherwise needs to be rewritten to accumulate the Boolean result as a parameter of \( \overline{\mathcal{O}_K}^* \).

5. Relaxing contexts for delimited continuations

In section 3 we showed how to get a pushdown abstraction by separating continuations from the heap that stores values. This separation breaks down when continuations themselves become values via first-class control operators. The glaring issue is that continuations become “storeable” and relevant to the execution of functions.

| (grant \( P e, \rho, \sigma, \kappa \)) | (e, \rho, \sigma, \kappa | \( P \mapsto \text{Grant} \)) |
| (frame \( P e, \rho, \sigma, \kappa \)) | (e, \rho, \sigma, \kappa | \( P \mapsto \text{Deny} \)) |
| (test \( P e_0 e_1, \rho, \sigma, \kappa \)) | (e_0, \rho, \sigma, \kappa \text{ if } True = \overline{\mathcal{O}_K}(P, \kappa) |
| (\{ e_1, \rho, \sigma, \kappa \text{ if } False = \overline{\mathcal{O}_K}(P, \kappa) |

\[
\text{Figure 8. CM machine semantics}
\]

The frame construct ensures that only a given set of permissions are allowed, but not necessarily granted. The security is in the semantics of test: we can test if all marks in some set \( P \) have been granted in the stack without first being denied; this involves crawling the stack:

\[
\overline{\mathcal{O}_K}(\phi, \kappa) = True
\]

\[
\overline{\mathcal{O}_K}(P, \epsilon^m) = \text{pass}(P, m)
\]

\[
\overline{\mathcal{O}_K}(P, \phi^m \kappa) = \text{pass}(P, m) \text{ and } \overline{\mathcal{O}_K}(P \setminus m^{-1}(\text{Grant}), \kappa)
\]

where \( \text{pass}(P, m) = P \cap m^{-1}(\text{Deny}) \).
But, it was precisely the irrelevance that allowed the separation of $\sigma$ and $\Xi$. Specifically, the store components of continuations become elements of the store’s codomain — a recursion that can lead to an unbounded state space and therefore a non-terminating analysis. We apply the AAM methodology to cut out the recursion; whenever a continuation is captured to go into the store, we allocate an address to approximate the store component of the continuation.

We introduce a new environment, $\chi$, that maps these addresses to the stores they represent. The stores that contain addresses in $\chi$ are then open, and must be paired with $\chi$ to be closed. This poses the same problem as before with contexts in storeable continuations. Therefore, we give up some precision to regain termination by flattening these environments when we capture continuations. Fresh allocation still maintains the concrete semantics, but we necessarily lose some ability to distinguish contexts in the abstract.

5.1 Case study of first-class control: shift and reset

We choose to study shift and reset [9] because delimited continuations have proven useful for implementing web servers [22, 21], providing processes isolation in operating systems [21], representing computational effects [13], modularly implementing error-correcting parsers [12], and finally undeclared continuations are passé for good reason [20]. Even with all their uses, however, their semantics can yield control-flow possibilities that surprise their users. A precise static analysis that illuminates their behavior is then a valuable tool.

Our concrete test subject is the abstract machine for shift and reset adapted from Biernacki et al. [2] in the “eval, continue” style in [Figure 9]. The figure elides the rules for standard function calls. The new additions to the state space are a new kind of value, $\text{comp}(k)$, and a meta-continuation, $C \in M\text{Kont} = \text{Kont}^*$ for separating continuations by their different prompts. Composable continuations are indistinguishable from functions, so even though the meta-continuation is concretely a list of continuations, its conses are notated as function composition: $\kappa \circ C$.

5.2 Reformulated with continuation stores

The machine in [Figure 9] is transformed now to have three new tables: one for continuations, one as discussed in the section beginning to close store continuations, and one for meta-continuations. The first is like previous sections, albeit continuations may now have the approximate form that is storeable. The meta-continuation table is more like previous sections because meta-contexts are not storeable. Meta-continuations do not have simple syntactic strategies for bounding their size, so we choose to bound them to size 0. They could be paired with lists of $\text{Kont}$ bounded at an arbitrary $n \in \mathbb{N}$, but we simplify for presentation.

Contexts for continuations are still at function application, but now contain the $\chi$. Contexts for meta-continuations are in two places: manual prompt introduction via $\text{reset}$, or via continuation invocation. At continuation capture time, continuation contexts are approximated to remove $\hat{\sigma}$ and $\chi$ components. The different context
Again we have a metafunction \( \text{pop} \), but this time to interpret approximated continuations:

\[
\text{pop}(\Xi, x, \hat{k}) = \text{pop}^* (\hat{k}, \varnothing)
\]

where \( \text{pop}^* (e, G) = \emptyset \)

\[
\text{pop}^* (\phi, G) = \{ (\phi, \tau) \}
\]

\[
\text{pop}^* (\tau, G) = \bigcup_{\iota \in G'} (\text{pop}^* (\iota, G \cup G'))
\]

where \( G' = \bigcup_{\iota \in I(\tau)} \Xi_\iota (\tau) \setminus G \)

\[
I(\tau) = (\tau)
\]

\[
I((e, \rho, a)) = \{ (e, \rho, \hat{\sigma}, \chi) \in \text{dom}(\Xi_\iota) : \hat{\sigma} \in \chi(a), \chi' \subseteq \chi \}
\]

Notice that since we flatten \( \chi' \)s together, we need to compare for containment rather than for equality (in \( I \)). A variant of this semantics with GC is available in the PLT redex models.

**Comparison to CPS transform to remove shift and reset:** We lose precision if we use a CPS transform to compile away shift and reset forms, because variables are treated less precisely than continuations. Consider the following program and its CPS transform for comparison:

\[
\begin{align*}
\text{let*} & \quad \text{for} (x: x) \quad \text{do}\quad \\
& \quad \{ (\lambda (y) \text{ shift } k (k \text{ y})))
\end{align*}
\]

\[
\{ \text{g} (0 \text{ g 1))})
\]

\[
\begin{align*}
\text{let*} & \quad \text{for} (x: x) \quad \text{do}\quad \\
& \quad \{ (\lambda (x k) \text{ x}))
\end{align*}
\]

\[
\{ \text{f} (\lambda (y) \text{ f}) (j (j \text{ y}))
\]

\[
\{ \text{g} (\lambda (x) \text{ h})
\]

\[
\{ \text{h} (f \text{ z} (\lambda \text{ f v))}
\]

\[
\{ \text{id v (\lambda (1) 1)))}
\]

\[
\{ \text{g} (0 \text{ g 0 v} (g 1 \text{ g 1 v)} (\text{g g 0 v} g 1)))
\]

The CESK\( _\Xi \) machine with a monovariant allocation strategy will predict the CPS\( _\Xi \)'s version returns true or false. In analysis literature, “monovariant” means variables get one address, namely themselves. Our specialized analysis for delimited control will predict the non-CPS\( _\Xi \)'s version returns true.

### 6. Short-circuiting via “summarization”

All the semantics of previous sections have a performance weakness that many analyses share: unnecessary propagation. Consider two portions of a program that do not affect one another’s behavior. Both can change the store, and the semantics will be unaware that the changes will not interfere with the other’s execution. The more possible stores there are in execution, the more possible contexts in which a function will be evaluated. Multiple independent portions of a program may be reused with the same arguments and store contents they depend on, but changes to irrelevant parts of the store lead to redundant computation. The idea of skipping from a change past several otherwise unchanged states to uses of the change is called “sparseness” in the literature [25][28][38].

Memoization is a specialized instance of sparseness; the base stack may change, but the evaluation of the function does not, so given an already computed result we can jump straight to the answer. We use the vocabulary of “relevance” and “irrelevance” so that future work can adopt the ideas of sparseness to reuse contexts in more ways.

Recall the context irrelevance lemma: if we have seen the results of a computation before from a different context, we can reuse them. The semantic counterpart to this idea is a memo table that we extend when popping and appeal to when about to push. This simple idea works well with a deterministic semantics, but the nondeterminism of abstraction requires care. In particular, memo table entries can end up mapping to multiple results, but not all results

\[
\zeta, \Xi, M \leftrightarrow \zeta', \Xi', M'
\]

\[
\{ (e_0 \ e_1), \rho, \sigma, \text{appL}(e_1, \rho) : \Xi, M \}
\]

\[
\{ (e_0, \rho, \sigma, \text{appR}(\lambda x. e, \rho) : \Xi, M \}
\]

will be found at the same time. Note the memo table space:

\[
M \in \text{Memo} \rightarrow \rho (\text{Relevant})
\]

Relevant ::= \( \langle e, \rho, \sigma \rangle \)

There are a few ways to deal with multiple results:

1. rerun the analysis with the last memo table until the table doesn’t change (expensive).
2. short-circuit to the answer but also continue evaluating anyway (negates most benefit of short-circuiting), or
3. use a frontier-based semantics like in subsection 3.3 with global \( \Xi \) and \( M \), taking care to
   
   (a) at memo-use time, still extend \( \Xi \) so later memo table extensions will “flow” to previous memo table uses, and
   
   (b) when \( \Xi \) and \( M \) are extended at the same context at the same time, also create states that act like the \( M \) extension point also returned to the new continuations stored in \( \Xi \).

We will only discuss the final approach. The same result can be achieved with a one-state-at-a-time frontier semantics, but we believe this is cleaner and more parallelizable. Its second subpoint we will call the “push/pop rendezvous.” The rendezvous is necessary because there may be no later push or pop steps that would regularly appeal to either (then extended) table at the same context. The frontier-based semantics then makes sure these pushes and pops find each other to continue on evaluating. In pushdown and nested word automata literature, the push to pop short-circuiting step is called a “summary edge” or with respect to the entire technique, “summarization.” We find the memoization analogy appeals to programmers’ and semanticians’ operational intuitions.

A second concern for using memo tables is soundness. Without the completeness property of the semantics, memoized results in, e.g., an inexactly GC’d machine, can have dangling addresses since the possible stacks may have grown to include addresses that were previously garbage. These addresses would not be garbage at first, since they must be mapped in the store for the contexts to coincide, but during the function evaluation the addresses can become garbage. If they are supposed to then be live, and are used (presumably they are reallocated post-collection), the analysis will miss paths it must explore for soundness.

**The rules in Figure 11 are the importantly changed rules from section 3 that short-circuit to memoized results.** The technique looks more like memoization with a CESK\( _\Xi \) machine, since the memoization points are truly at function call and return boundaries. The pop function would need to also update \( M \) if it dereferences through a context, but otherwise the semantics are updated mutatis mutandis.
where

\[ I = \bigcup_{\chi \in \mathcal{F}} \{ (\langle s, \chi \rangle, \Xi', M' \} : s, \Xi, M \rightarrow \chi, \Xi', M' \} \]

\[ R' = \pi_2 I \]

\[ \Delta \Xi' = \Xi' \cap \Xi \]

\[ M' = M' \cap M \]

\[ F' = \pi_1 R' \cup \{ (q, \rho, \pi, \kappa) : \tau \in \text{dom}(\Delta \Xi) \cap \text{dom}(\Delta M). \}

\[ \kappa \in \Delta \Xi(\tau), \langle q, \rho, \pi \rangle \in \Delta M(\tau) \} \]

\[
F(I, R, F, \Xi, M) = (S \cup F, R \cup R', F' \setminus S, \Xi', M')
\]

7. Related Work

The immediately related work is that of PDCFA [10] [11], CFA2 [36] [37], and AAM [33]. The stack frames in CFA2 that boost precision are an orthogonal feature that fit into our model as an irrelevant component along with the stack, which we did not cover in detail due to space constraints. The version of CFA2 that handles call/cc does not handle composable control, is dependent on a restricted CPS representation, and has untunable precision for first-class continuations. Our semantics adapts to a restricted CPS representation, and has untunable precision for does not handle composable control, is dependent on component along with the stack, which we did not cover in de-collection [11] also fits into our model (subsection 4.1’s moving the meta-continuation operations, and thus this work supersedes theirs; the machinery is in fact a strict generalization. The extended version of PDCFA that inspects the stack to do garbage collection [11] also fits into our model (subsection 4.1’s Γ). We did additional work to improve the performance of the AAM approach in Johnson et al. [16] that can almost entirely be imported for the work in this paper. The technique that does not apply is “store counting” for lean representation of the stack component of states when there is a global abstract store, an assumption that does not hold for garbage collection. The implementation [2] has pushdown modules that use the ideas in this paper.

Stack inspection Stack inspecting flow analyses also exist, but operate on pre-constructed regular control-flow graphs [1], so the CFGs cannot be trimmed due to the extra information at construction time, leading to less precision. Backward analyses for stack inspection also exist, with the same prerequisite [4].

Pushdown models and memoization The idea of relating pushdown automata with memoization is not new. In 1971, Stephen Cook [6] devised a transformation to simulate 2-way (on a fixed input) deterministic pushdown automata in time linear in the size of the input, that uses the same “context irrelevance” idea to skip from state q seen before to a corresponding first state that pops the stack below where q started (called a terminator state). Six years later, Neil D. Jones [13] simplified the transformation instead to interpret a stack machine program to work on-the-fly still on a deterministic machine, but with the same idea of using memo tables to remember corresponding terminator states. Thirty-six years after that, at David Schmidt’s Festschrift, Robert Glück extended the technique to 2-way non-deterministic pushdown automata, and showed that the technique can be used to recognize context-free languages in the standard \(O(n^3)\) time [14]. Glück’s technique (as yet, correctness unproven) uses the meta-language’s stack with a deeply recursive interpretation function to preclude the use of a frontier and something akin to \(http://github.com/dvanhorn/oam\). By exploring the state space depth-first, the interpreter can find all the different terminators a state can reach one-by-one by destructively updating the memo table with the “latest” terminator found. The trade-offs with this technique are that it does not obviously scale to first-class control, and the search can overflow the stack when interpreting moderate-sized programs. A minor disadvantage is that it is also not a fair evaluation strategy when allocation is unbounded. The technique can nevertheless be a viable alternative for languages with simple control-flow mechanisms. It has close similarities to “Big-CFA2” in Vardoulakis’ dissertation [33].

Analysis of pushdown automata Pushdown models have existed in the first-order static analysis literature [25] Chapter 7 [29], and the first-order model checking literature [1], for some time. These methods already assume a pushdown model as input, and constructing a model from a first-order program is trivial. In the setting of higher-order functions and first-class control, model construction is an additional problem – the one we solve here. Additionally, the algorithms employed in these works expect a complete description of the model up front, rather than work with a modified step function (also called post), such as in “on-the-fly” model-checking algorithms for finite state systems [33].

Derivation from abstract machines 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 [30]. The expressiveness of CPDAs outweighs that of PDAs, but it is unclear how to adapt higher-order recursion schemes (HORS) 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.

8. Conclusion

As the programming world continues to embrace behavioral values like functions and continuations, it becomes more important to import the powerful techniques pioneered by the first-order analysis and model-checking communities. It is our view that systematic approaches to analysis construction are pivotal to scaling them to production programming languages. We showed how to systematically construct executable concrete semantics that point-wise abstract to pushdown analyses of higher-order languages. We bypass the automata theoretic approach so that we are not chained to a pushdown automaton to model features such as first-class composable control operators. The techniques employed for pushdown analysis generalize gracefully to apply to non-pushdown models and give better precision than regular methods.

References


