Abstracting Abstract Control

J. Ian Johnson
Northeastern University
ianj@ccs.neu.edu

David Van Horn
University of Maryland
dvanhorn@cs.umd.edu

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 few, 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 control stack.

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 [7, 39]. 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 [8]). This alternative, called abstracting abstract machines (AAM) [34], is a simple idea that is generally applicable to even the most dynamic of languages, *e.g.*, JavaScript [19]. 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 (§2) and then revising the derivation to incorporate an exact representation of the control stack (§3). We then show how the approach scales to stack-reflecting language features such as garbage collection and stack inspection (§4), and stack-reifying features in the form of first-class delimited control operators (§5). 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 [17, 36].

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 toolbelt, we hope this work will help the community work towards technical feats like Astrée [7]. 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 [33]. 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 $CESK_t^*$ machine *schema*

The case studies in this paper have full implementations in PLT redex [12] available online.[1] They all build off the call-by-value untyped lambda calculus, whose semantics we recall in small-step reduction semantics style:

$$\begin{array}{rl} & E[(\lambda x.e\ v)] \longmapsto_\beta E[[v/x]e] \\ \text{where} & E \in \mathit{EvaluationContext} ::= [\,] \mid (E\ e) \mid (v\ E) \\ & e \in \mathit{Expr} ::= x \mid (e\ e) \mid v \qquad v \in \mathit{Value} ::= \lambda x.e \end{array}$$

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 $\longmapsto_\beta$), 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.[2] The environment and store work together to represent ongoing substitutions, and the continuation represents where 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 a is "fresh" if $a \notin \mathrm{dom}(\sigma)$, but this is just a specification, not an implementation. Concretely, there is an allocation function $\mathit{alloc} : \mathit{CESK} \to \mathit{Addr}$ that will produce an address for the machine to use. We use a slightly non-standard CESK machine that allows alloc to produce arbitrary addresses instead of just fresh ones.

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 presentation are in the use of $\in$ instead of $=$ to make lookups non-deterministic, and weak updates with $\sqcup$ instead of strong updates to allow sound reuse of addresses:

$$\begin{aligned} \sigma \sqcup [a \mapsto v] &= \sigma[a \mapsto \sigma(a) \sqcup v] \\ v \sqcup v' &= \{v, v'\} \\ \{v, \ldots\} \sqcup v' &= \{v, \ldots, v'\} \end{aligned}$$

$$\begin{array}{rlrl} \varsigma \in \mathit{CESK} & ::= \langle e, \rho, \sigma, \kappa \rangle & & \\ \ell \in \mathit{Lam} & ::= \lambda x.e & v \in \mathit{Value} & ::= (\ell, \rho) \\ \rho \in \mathit{Env} & = \mathit{Var} \rightharpoonup \mathit{Addr} & \sigma \in \mathit{Store} & = \mathit{Addr} \rightharpoonup \wp(\mathit{Value}) \\ \phi \in \mathit{Frame} & ::= \mathbf{appL}(e, \rho) \mid \mathbf{appR}(v) & & \\ \kappa \in \mathit{Kont} & = \mathit{Frame}^* & & \\ x \in \mathit{Var} & \text{a set} & a, b \in \mathit{Addr} & \text{a set} \\ \mathit{alloc} & : \mathit{CESK} \to \mathit{Addr} & & \end{array}$$

Figure 1. CESK semantic spaces

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

Figure 2. CESK machine

$$\begin{array}{rlrl} \hat{\varsigma} \in \widehat{\mathit{CESK}}_t & ::= \langle e, \rho, \hat{\sigma}, \hat{\kappa} \rangle_t & & \\ \hat{\sigma} \in \widehat{\mathit{Store}} & = \mathit{Addr} \to \wp(\mathit{Storeable}) & \hat{\kappa} \in \widehat{\mathit{Kont}} & ::= \epsilon \mid \phi{:}a \\ t, u \in \mathit{Time} & & \mathit{Storeable} & ::= \hat{\kappa} \mid v \end{array}$$

Figure 3. $CESK_t^*$ semantic spaces

Removing recursion: In order to make $\longmapsto$ 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 cons is 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 $\mathit{alloc} : \mathit{State} \to \mathit{Addr}$ 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 [15] suggests less arbitrary constructions, and to think of Time as a space of abstract traces, though the "abstraction" need not be sound [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 $CESK_t^*$ machine. The semantics of this machine follow the weak update and non-deterministic lookup principles of AAM in Figure 4.

If we run the $CESK_t^*$ 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

[1] `http://github.com/dvanhorn/aac`

[2] 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.

$$\begin{array}{c}
\hat\varsigma \longmapsto \hat\varsigma' \quad a = alloc(\hat\varsigma) \quad u = tick(\hat\varsigma) \\
\hline
\begin{array}{r|l}
\langle x, \rho, \hat\sigma, \hat\kappa\rangle_t & \langle v, \hat\sigma, \hat\kappa\rangle_u \text{ if } v \in \sigma(\rho(x)) \\
\langle (e_0\ e_1), \rho, \hat\sigma, \hat\kappa\rangle_t & \langle e_0, \rho, \hat\sigma', \mathbf{appL}(e_1, \rho){:}a\rangle_u \\
\text{where} & \hat\sigma' = \hat\sigma \sqcup [a \mapsto \hat\kappa] \\
\langle v, \rho, \hat\sigma, \mathbf{appL}(e, \rho'){:}b\rangle_t & \langle e, \rho', \hat\sigma, \mathbf{appR}(v, \rho){:}b\rangle_u \\
\langle v, \hat\sigma, \mathbf{appR}(\lambda x.e, \rho){:}b\rangle_t & \langle e, \rho', \hat\sigma', \hat\kappa\rangle_u \text{ if } \hat\kappa \in \hat\sigma(b) \\
\text{where} & \rho' = \rho[x \mapsto a] \\
 & \hat\sigma' = \hat\sigma \sqcup [a \mapsto v]
\end{array}
\end{array}$$

Figure 4. $CESK_t^*$ semantics

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 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 Ξ, 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:

$$\begin{aligned}
\widehat{State} &= \widehat{CESK}_t \times KStore \\
Storeable &= Value \\
\kappa \in Kont &::= \epsilon \mid \phi{:}\tau \\
\tau \in Context &::= \langle e, \rho, \sigma\rangle_t \\
\Xi \in KStore &= Context \rightharpoonup \wp(Kont)
\end{aligned}$$

The semantics is modified slightly in Figure 5 to use Ξ instead of σ 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 $\longmapsto$ 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 $alloc$ can have an unbounded codomain. The correctness claim is therefore a correspondence between the same machine but with an unbounded stack, no Ξ, and $alloc$, $tick$ functions that behave the same disregarding the different representations (a reasonable assumption).

$$\begin{array}{c}
\hat\varsigma, \Xi \longmapsto \hat\varsigma', \Xi' \quad a = alloc(\hat\varsigma, \Xi) \quad u = tick(\hat\varsigma, \Xi) \\
\hline
\begin{array}{r|l}
\langle x, \rho, \sigma, \hat\kappa\rangle_t, \Xi & \langle v, \sigma, \hat\kappa\rangle_u, \Xi \text{ if } v \in \sigma(\rho(x)) \\
\langle (e_0\ e_1), \rho, \sigma, \hat\kappa\rangle_t, \Xi & \langle e_0, \rho, \sigma, \mathbf{appL}(e_1, \rho){:}\tau\rangle_u, \Xi' \\
\text{where} & \tau = \langle (e_0\ e_1), \rho, \sigma\rangle_t \\
 & \Xi' = \Xi \sqcup [\tau \mapsto \hat\kappa] \\
\langle v, \sigma, \mathbf{appL}(e, \rho'){:}\tau\rangle_t, \Xi & \langle e, \rho', \sigma, \mathbf{appR}(v){:}\tau\rangle_u, \Xi \\
\langle v, \rho, \sigma, \mathbf{appR}(\lambda x.e, \rho'){:}\tau\rangle_t, \Xi & \langle e, \rho'', \sigma', \hat\kappa\rangle_u, \Xi \text{ if } \hat\kappa \in \Xi(\tau) \\
\text{where} & \rho'' = \rho'[x \mapsto a] \\
 & \sigma' = \sigma \sqcup [a \mapsto v]
\end{array}
\end{array}$$

Figure 5. $CESK_t^*\,\mathit{\Xi}$ semantics

3.1 Correctness

The high level argument for correctness exploits properties of both machines. Where the stack is unbounded (call this $CESK_t$), if every state in a trace shares a common tail in their continuations, that tail is *irrelevant*. This means the tail can be replaced with anything and still produce a valid trace. We call this property more generally, "context irrelevance." The $CESK_t^*\,\mathit{\Xi}$ machine maintains an invariant on Ξ that says that $\hat\kappa \in \Xi(\tau)$ represents a trace in $CESK_t$ that starts at the base of $\hat\kappa$ and reaches τ with $\hat\kappa$ on top. We can use this invariant and context irrelevance to translate steps in the $CESK_t^*\,\mathit{\Xi}$ machine into steps in $CESK_t$. The other way around, we use a proposition that a full stack is represented by Ξ via unrolling and follow a simple simulation argument.

The common tail proposition we will call ht and the replacement function we will call rt; they both have obvious inductive and recursive definitions respectively. The invariant is stated with respect to the entire program, e_{pgm}:

$$\frac{}{inv_\Xi(\bot)} \qquad \frac{\begin{array}{c} inv_\Xi(\Xi) \\ \forall \hat\kappa_c \in K.base(\hat\kappa_c) \longmapsto^*_{CESK_t} \langle e_c, \rho_c, \sigma_c, \kappa_c{+}{+}\epsilon\rangle_{t_c}\end{array}}{inv_\Xi(\Xi[\langle e_c, \rho_c, \sigma_c\rangle_{t_c} \mapsto K])}$$

$$\frac{base(\hat\kappa) \longmapsto^*_{CESK_t} \langle e, \rho, \sigma, \hat\kappa{+}{+}\epsilon\rangle_t \qquad inv_\Xi(\Xi)}{inv(\langle e, \rho, \sigma, \hat\kappa\rangle_t, \Xi)}$$

where

$$\begin{aligned}
base(\epsilon) &= \langle e_{pgm}, \bot, \bot, \epsilon\rangle_{t_0} \\
base(\phi{:}\langle e_c, \rho_c, \sigma_c\rangle_{t_c}) &= \langle e_c, \rho_c, \sigma_c, \epsilon\rangle_{t_c}
\end{aligned}$$

We use $\cdot{+}{+}\epsilon$ to treat τ like ϵ and construct a continuation in $Kont$ rather than $\widehat{Kont}$.

Lemma 1 (Context irrelevance)**.** *For all traces $\pi \in {CESK_t}^*$ and continuations κ such that $ht(\pi, \kappa)$, for any κ', $rt(\pi, \kappa, \kappa')$ is a valid trace.*

Proof. Simple induction on π and cases on $\longmapsto_{CESK_t}$. □

Lemma 2 ($CESK_t^*\,\mathit{\Xi}$ Invariant)**.** *For all $\varsigma, \varsigma' \in \widehat{State}$, if $inv(\varsigma)$ and $\varsigma \longmapsto \varsigma'$, then $inv(\varsigma')$*

Proof. Routine case analysis. □

Note that the injection of e_{pgm} into $CESK_t^*\,\mathit{\Xi}$, $(\langle e_{pgm}, \bot, \bot, \epsilon\rangle_{t_0}, \bot)$, trivially satisfies inv.

The unrolling proposition is the following

$$\frac{}{\epsilon \in unroll_\Xi(\epsilon)} \qquad \frac{\hat\kappa \in \Xi(\tau), \kappa \in unroll_\Xi(\hat\kappa)}{\phi{:}\kappa \in unroll_\Xi(\phi{:}\tau)}$$

Theorem 3 (Correctness)**.** *For all expressions e_{pgm},*

- **Soundness:** *if $\varsigma \longmapsto_{CESK_t} \varsigma'$, $inv(\varsigma\{\kappa := \hat\kappa\}, \Xi)$, and $\kappa \in unroll_\Xi(\hat\kappa)$, then there are $\Xi', \hat\kappa'$ such that $\varsigma\{\kappa := \hat\kappa\}, \Xi \longmapsto_{CESK_t^*\,\mathit{\Xi}} \varsigma'\{\kappa := \hat\kappa'\}, \Xi'$ and $\kappa' \in unroll_{\Xi'}(\hat\kappa')$*

$$\hat{\varsigma} \in \widehat{CESIK} = \langle e, \rho, \sigma, \iota, \hat{\kappa} \rangle \qquad \begin{array}{r}\iota \in LKont = Frame^* \\ \hat{\kappa} \in Kont ::= \epsilon \mid \tau\end{array}$$

Figure 6. $CESIK^*\Xi$ semantic spaces

- **Completeness:** *if* $\hat{\varsigma}, \Xi \longmapsto_{CESK_t^*\Xi} \hat{\varsigma}', \Xi'$ *and* $inv(\hat{\varsigma}, \Xi)$*, for all* κ*, if* $\kappa \in unroll_\Xi(\hat{\varsigma}.\hat{\kappa})$ *then there is a* κ' *such that* $\hat{\varsigma}\{\hat{\kappa} := \kappa\} \longmapsto_{CESK_t} \hat{\varsigma}'\{\hat{\kappa} := \kappa'\}$ *and* $\kappa' \in unroll_\Xi(\hat{\varsigma}'.\hat{\kappa})$.

3.2 Engineered semantics for efficiency

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

1. Observe that Ξ 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 in 2 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 [2] 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."[3]

The resulting shuffling of the semantics to accommodate this new representation is in Figure 7. The extension to Ξ 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 Ξ once ι is empty, or to use a helper metafunction to describe a "pop" of both ι and κ. 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:

$$\begin{aligned} pop(\phi{:}\iota, \hat{\kappa}, \Xi) &= \{(\phi, \iota, \hat{\kappa})\} \\ pop(\epsilon, \tau, \Xi) &= \{(\phi, \iota, \hat{\kappa}) \;:\; (\phi{:}\iota, \hat{\kappa}) \in \Xi(\tau)\} \end{aligned}$$

However, tail calls make the dereferenced τ lead to (ϵ, τ'). Because abstraction makes the store grow monotonically in a finite space, it's possible that $\tau' = \tau$ and a naive recursive definition of pop will diverge chasing these contexts. Now pop must save all the contexts it dereferences in order to guard against divergence. So

$$\hat{\varsigma}, \Xi \longmapsto \hat{\varsigma}', \Xi' \qquad a = alloc(\hat{\varsigma}, \Xi)$$

$$\begin{array}{r|l}
\langle x, \rho, \sigma, \iota, \hat{\kappa} \rangle, \Xi & \langle v, \sigma, \iota, \hat{\kappa} \rangle, \Xi \text{ if } v \in \sigma(\rho(x)) \\
\langle (e_0\ e_1), \rho, \sigma, \iota, \hat{\kappa} \rangle, \Xi & \langle e_0, \rho, \sigma, \mathbf{appL}(e_1, \rho){:}\iota, \hat{\kappa} \rangle, \Xi \\
\langle v, \sigma, \iota, \hat{\kappa} \rangle, \Xi & \langle e, \rho', \sigma, \mathbf{appR}(v, \rho){:}\iota', \hat{\kappa}' \rangle, \Xi \\
 & \text{if } \mathbf{appL}(e, \rho'), \iota', \hat{\kappa}' \in pop(\iota, \hat{\kappa}, \Xi) \\
\langle v, \sigma, \iota, \hat{\kappa} \rangle, \Xi & \langle e, \rho[x \mapsto a], \sigma', \epsilon, \tau \rangle, \Xi' \\
 & \text{if } \mathbf{appR}(\lambda x.e, \rho), \iota', \hat{\kappa}' \in pop(\iota, \hat{\kappa}, \Xi) \\
\text{where} & \sigma' = \sigma \sqcup [a \mapsto v] \\
 & \tau = (\langle \lambda x.e, \rho \rangle, v, \sigma) \\
 & \Xi' = \Xi \sqcup [\tau \mapsto (\iota, \hat{\kappa})]
\end{array}$$

Figure 7. $CESIK^*\Xi$ semantics

$$\begin{aligned} pop(\iota, \hat{\kappa}, \Xi) &= pop^*(\iota, \hat{\kappa}, \Xi, \varnothing) \text{ where} \\ pop^*(\epsilon, \epsilon, \Xi, G) &= \varnothing \\ pop^*(\phi{:}\iota, \hat{\kappa}, \Xi, G) &= \{(\phi, \iota, \hat{\kappa})\} \\ pop^*(\epsilon, \tau, \Xi, G) &= \{(\phi, \iota, \hat{\kappa}) \;:\; (\phi{:}\iota, \hat{\kappa}) \in \Xi(\tau)\} \\ &\quad \cup \bigcup_{\tau' \in G'} pop^*(\epsilon, \tau', \Xi, G \cup G') \\ &\text{where } G' = \{\tau' \;:\; (\epsilon, \tau') \in \Xi(\tau)\} \setminus G \end{aligned}$$

In practice, one would not expect G to grow very large. Had we chosen the first strategy, the issue of divergence is delegated to the machinery from the fixed-point computation.[4] However, when adding the administrative state, the "seen" check requires searching a far larger set than we would expect G to be.

$$\begin{aligned} \mathcal{F}_e(S, R, F, \Xi) &= (S \cup F, R \cup R', F' \setminus S, \Xi') \\ I &= \bigcup_{s = (\hat{\varsigma}, \sigma) \in F} \{(\langle \hat{\varsigma}, \hat{\varsigma}' \rangle, \Xi') \;:\; \hat{\varsigma}, \Xi \longmapsto \hat{\varsigma}', \Xi'\} \\ R' &= \pi_0 I \qquad F' = \pi_1 R' \qquad \Xi' = \bigsqcup \pi_1 I \end{aligned}$$

For a program e, we will say $(\varnothing, \varnothing, \{\langle e, \bot, \bot, \epsilon, \epsilon \rangle\}, \bot)$ is the bottom element of $\mathcal{F}_e$'s domain. The "analysis" then is then the pair of the R and Ξ components of $\mathbf{lfp}(\mathcal{F}_e)$.

Correctness The correctness argument for this semantics is not about single steps but instead about the entire relation that $\mathcal{F}$ computes. The argument is that the R and Ξ components of the system represent a slice of the unbounded relation $\longmapsto_{CESK}$ (restricted to reachable states). We will show that traces in any $n \in \mathbb{N}$ times we *unfold* $\longmapsto_{CESK}$ from the initial state, there is a corresponding m applications of $\mathcal{F}$ that reify into a relation that exhibit the same trace. Conversely, any trace in the reification of $\mathcal{F}_e^m(\bot)$ has the same trace in some n unfoldings of $\longmapsto_{CESK}$. For an arbitrary $alloc$ function, we cannot expect $\mathcal{F}$ to have a fixed point, so this property is the best we can get. For a finite $alloc$ function, Kleene's fixed point theorem dictates there is a m such that $\mathcal{F}_e^m(\bot)$ is a fixed point, so every trace in the reified relation is also a trace in an unbounded number of unfoldings of $\longmapsto_{CESK}$. This is the corresponding completeness argument for the algorithm.

$$\begin{aligned} unfold(\varsigma_0, \longmapsto, 0) &= \{(\varsigma_0, \varsigma) \;:\; \varsigma_0 \longmapsto \varsigma\} \\ unfold(\varsigma_0, \longmapsto, n+1) &= unfold_1(unfold(\varsigma_0, \longmapsto, n)) \\ \textit{where } unfold_1(R) &= R \cup \{(\varsigma, \varsigma') \;:\; (\_, \varsigma) \in R, \varsigma \longmapsto \varsigma'\} \end{aligned}$$

The reification simply realizes all possible complete continuations that a state could have, given Ξ:

$$\frac{\langle \langle e, \rho, \sigma, \hat{\kappa} \rangle, \langle e', \rho', \sigma', \hat{\kappa}' \rangle \rangle \in R \qquad \kappa \in tails_\Xi(\hat{\kappa})}{\langle e, \rho, \sigma, \hat{\kappa} {+\!+} \kappa \rangle \longmapsto_{reify(S,R,F,\Xi)} \langle e', \rho', \sigma' \hat{\kappa}' {+\!+} \kappa \rangle}$$

[3] Since the continuation is either ϵ or a context, CFA2 calls these "entries" to mean execution entry into the program (ϵ) or a function (τ). One can also understand these as entries in a table (Ξ). We stay with the "continuation" nomenclature because they represent full continuations.

[4] CFA2 employs the first strategy and calls it "transitive summaries."

The additional judgment about tails is a variant of *unroll* in order to get a common tail:

$$\frac{}{\epsilon \in \mathit{tails}_{\Xi}(\epsilon)} \qquad \frac{\hat{\kappa} \in \Xi(\tau) \qquad \kappa \in \mathit{unroll}_{\Xi}(\hat{\kappa})}{\kappa \in \mathit{tails}_{\Xi}(\phi{:}\tau)}$$

Theorem 4 (Correctness)**.** *For all e_0, let $\varsigma_0 = \langle e_0, \bot, \bot, \epsilon\rangle$ in $\forall n \in \mathbb{N}, \varsigma, \varsigma' \in \mathit{CESK}$:*

- *if $(\varsigma, \varsigma') \in \mathit{unfold}(\varsigma_0, \longmapsto_{\mathit{CESK}}, n)$ then there is an m such that $\varsigma \longmapsto_{\mathit{reify}(\mathcal{F}^m_{e_0}(\bot))} \varsigma'$*
- *if $\varsigma \longmapsto_{\mathit{reify}(\mathcal{F}^n_{e_0}(\bot))} \varsigma'$ then there is an m such that $(\varsigma, \varsigma') \in \mathit{unfold}(\varsigma_0, \longmapsto_{\mathit{CESK}}, m)$*

Proof. By induction on n. □

3.3 Remarks about complexity

The common tradeoff for performance over precision is to use a global store. A representation win originally exploited by Shivers [**?**] is to represent the seen states' stores by the *age* of the store. A context in this case contains the store age for faster comparison. Old stores are mostly useless, so a useful representation for the seen set is as a map from the *rest of the state* to the store age. We will align with the analysis literature and call these "rest of state" objects, *points*. Note that since the store age becomes part of the state representation due to "context," there are considerably more points than in the comparable finite state approach. When we revisit a state because the store age is different from the last time we visited it (hence we're visiting a *new state*), we can clobber the old store age. A finite state approach may use less memory because the seen set will have a smaller domain (fewer distinctions made because of the lack of a "context" component), but it does not necessarily take less time. In our evaluation, we instead saw that although point counts were higher, visited states were lower in the pushdown setting.

Theoretically the state space is bigger, but in practice *The store age as part of the state representation can increase the state space considerably.* While this can indeed increase analysis time, There are two separate concepts that must be addressed when we see a spike in our metrics. We modified this approach to lessen the number of times the age must be updated to reduce the necessary propagations in our paper about optimizing abstract abstract machines [**?**]. More information about optimizing representations

4. Stack inspection and recursive metafunctions

Since we just showed how to produce a pushdown system from an abstract machine, some readers may be concerned that we have lost the ability to reason about the stack as a whole. This is not the case. The semantics may still refer to Ξ to make judgments about the possible stacks that can be realized at each state. The key is to interpret the functions making these judgments again with the AAM methodology.

Some semantic features allow a language to inspect some arbitrarily deep part of the stack, or compute a property of the whole stack before continuing. Java's access control security features are an example of the first form of inspection, and garbage collection is an example of the second. We will demonstrate both forms are simple first-order metafunctions that the AAM methodology will soundly interpret. Access control can be modeled with continuation marks, so we demonstrate with the CM machine of Clements and Felleisen.

Semantics that inspect the stack do so with metafunction calls that recur down the stack. Recursive metafunctions have a semantics 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 $\longmapsto$ to compute all terminal states (outputs):

$$\begin{aligned}
\mathit{terminal} &: \forall A.\text{relation } A \times A \to \wp(A) \\
\mathit{terminal}(\longmapsto, s) &= \mathit{terminal}^*(\varnothing, \{s\}, \varnothing)
\end{aligned}$$

$$\begin{aligned}
\text{where } \mathit{terminal}^*(S, \varnothing, T) &= T \\
\mathit{terminal}^*(S, F, T) &= \mathit{terminal}^*(S \cup F, F', T \cup T') \\
\text{where } T' &= \bigcup_{s \in F} \mathit{post}(s) \overset{?}{=} \varnothing \to \{s\}, \varnothing \\
F' &= \bigcup_{s \in F} \mathit{post}(s) \setminus S \\
\mathit{post}(s) &= \{s' \,:\, s \longmapsto s'\}
\end{aligned}$$

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, $\mathit{post}(s)$ will be empty, meaning s should be added to the terminal set T. We prove a correctness condition that allows us to reason equationally with $\mathit{terminal}$ later on:

Lemma 5 ($\mathit{terminal}^*$ correct)**.** *Fix $\longmapsto$. Constrain arbitrary S, F, T such that $T \sqsubseteq S$ and $\forall s \in T, \mathit{post}(s) = \varnothing$, $F \cap S = \varnothing$, and for all $s \in S$, $\mathit{post}(s) \subseteq S \cup F$.*

- ***Soundness:** for all $s \in S \cup F$, if $s \longmapsto^* s_t$ and $\mathit{post}(s_t) = \varnothing$ then $s_t \in \mathit{terminal}^*(S, F, T)$.*
- ***Completeness:** for all $s \in \mathit{terminal}^*(S, F, T)$ there is an $s_0 \in S \cup F$ such that $s_0 \longmapsto^* s$ and $\mathit{post}(s) = \varnothing$.*

Proof. By induction on $\mathit{terminal}^*$'s recursion scheme. □

Note that it is possible for metafunctions' rewrite rules to themselves use metafunctions, but the *seen* set (S) for *terminal* must be dynamically bound[5] – 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:

$$\begin{aligned}
\mathcal{KLL} &: \mathit{Frame}^* \to \wp(\mathit{Addr}) \\
\mathcal{KLL}(\hat{\kappa}) &= \mathcal{KLL}^*(\hat{\kappa}, \varnothing) \\
\mathcal{KLL}^*(\epsilon, L) &= L \\
\mathcal{KLL}^*(\phi{:}\hat{\kappa}, L) &= \mathcal{KLL}^*(\hat{\kappa}, L \cup \mathcal{T}(\phi)) \\
\text{where } \mathcal{T}(\mathbf{appL}(e, \rho)) &= \mathcal{T}(\mathbf{appR}(e, \rho)) = \mathcal{T}(e, \rho) \\
\mathcal{T}(e, \rho) &= \{\rho(x) \,:\, x \in \mathit{fv}(e)\}
\end{aligned}$$

[5] This is a reference to dynamic scope as opposed to lexical scope.

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

$$K\mathcal{LL}(\Xi, \hat{\kappa}) = \mathit{terminal}(\longmapsto, K\mathcal{LL}^*(\Xi, \hat{\kappa}, \varnothing))$$

$$K\mathcal{LL}^*(\Xi, \epsilon, L) \longmapsto L$$

$$K\mathcal{LL}^*(\Xi, \phi{:}\tau, L) \longmapsto K\mathcal{LL}^*(\Xi, \hat{\kappa}, L \cup \mathcal{T}(\phi)) \text{ if } \hat{\kappa} \in \Xi(\tau)$$

A garbage collecting semantics can choose to collect the heap with respect to each live set (call this Γ^*), or, soundly, collect with respect to their union (call this $\hat{\Gamma}$).[6] On the one hand we could have 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 σ. 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 $\mathcal{R}$:

$$\mathcal{R}(\mathit{root}, \sigma) = \{b : a \in \mathit{root}, a \leadsto_\sigma^* b\}$$

$$\frac{v \in \sigma(a) \qquad b \in \mathcal{T}(v)}{a \leadsto_\sigma b}$$

So the two collection methods are as follows. Exact GC produces different collected states based on the possible stacks' live addresses:[7]

$$\Gamma^*(\hat{\varsigma}, \Xi) = \{\hat{\varsigma}\{\sigma := \hat{\varsigma}.\sigma|_L\} : L \in \mathit{live}^*(\hat{\varsigma}, \Xi)\}$$

$$\mathit{live}^*(\langle e, \rho, \sigma, \hat{\kappa}\rangle, \Xi) = \{\mathcal{R}(\mathcal{T}(e, \rho) \cup L, \sigma) : L \in K\mathcal{LL}(\Xi, \hat{\kappa})\}$$

$$\frac{\hat{\varsigma}, \Xi \longmapsto \hat{\varsigma}', \Xi' \qquad \hat{\varsigma}' \in \Gamma^*(\varsigma', \Xi')}{\hat{\varsigma}, \Xi \longmapsto_{\Gamma^*} \hat{\varsigma}', \Xi}$$

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

$$\hat{\Gamma}(\hat{\varsigma}, \Xi) = \hat{\varsigma}\{\sigma := \hat{\varsigma}.\sigma|_{\widehat{\mathit{live}}(\hat{\varsigma}, \Xi)}\}$$

$$\widehat{\mathit{live}}(\langle e, \rho, \sigma, \hat{\kappa}\rangle, \Xi) = \mathcal{R}(\mathcal{T}(e, \rho) \cup \bigcup K\mathcal{LL}(\Xi, \hat{\kappa}), \sigma)$$

$$\frac{\hat{\varsigma}, \Xi \longmapsto \hat{\varsigma}', \Xi'}{\hat{\varsigma}, \Xi \longmapsto_{\hat{\Gamma}} \hat{\Gamma}(\varsigma', \Xi')\Xi'}$$

Without the continuation store, the baseline GC is

$$\Gamma(\hat{\varsigma}) = \hat{\varsigma}\{\sigma := \hat{\varsigma}.\sigma|_{\mathit{live}(\hat{\varsigma})}\}$$

$$\mathit{live}(e, \rho, \sigma, \kappa) = \mathcal{R}(\mathcal{T}(e, \rho) \cup K\mathcal{LL}(\hat{\kappa}), \sigma)$$

$$\frac{\hat{\varsigma} \longmapsto \hat{\varsigma}'}{\hat{\varsigma} \longmapsto_\Gamma \Gamma(\varsigma')}$$

Suppose at arbitrary times we decide to perform garbage collection rather than continue with garbage. So when $\hat{\varsigma} \longmapsto \hat{\varsigma}'$, we instead do $\hat{\varsigma} \longmapsto_\Gamma \hat{\varsigma}'$. The times we perform GC do not matter for soundness, since we are not analyzing GC behavior. However, garbage stands in the way of completeness. Mismatches in the GC application for the different semantics lead to mismatches in resulting state spaces, not just up to garbage in stores, but in spurious paths from dereferencing a reallocated address that was not first collected.

The state space compaction that continuation stores give us makes ensuring GC times match up for the completeness proposition tedious. Our statement of completeness then will assume both semantics perform garbage collection on every step.

Lemma 6 (Correctness of $K\mathcal{LL}$)**.** *For all $\Xi, \kappa, \hat{\kappa}, L$,*

- ***Soundness:*** *if $\kappa \in \mathit{unroll}_\Xi(\hat{\kappa})$ then $K\mathcal{LL}^*(\kappa, L) \in \mathit{terminal}(\longmapsto, K\mathcal{LL}^*(\Xi, \hat{\kappa}, L))$*
- ***Completeness:*** *for all $L' \in K\mathcal{LL}^*(\Xi, \hat{\kappa}, L)$ there is a $\kappa \in \mathit{unroll}_\Xi(\hat{\kappa})$ such that $L' = K\mathcal{LL}^*(\kappa, L)$.*

Proof. Soundness follows by induction on the unrolling. Completeness follows by induction on the trace from completeness in Lemma 5. □

Theorem 7 (Correctness of $\Gamma^*\, \mathit{CESK}_t^*\, \Xi$)**.** *For all expressions e_0,*

- **Soundness:** *if $\varsigma \longmapsto_{\Gamma\mathit{CESK}_t} \varsigma'$, $\mathit{inv}(\varsigma\{\kappa := \hat{\kappa}\}, \Xi)$, and $\varsigma.\kappa \in \mathit{unroll}_\Xi(\hat{\kappa})$, then there are $\Xi', \hat{\kappa}', \sigma'$ such that $\varsigma\{\kappa := \hat{\kappa}\}, \Xi \longmapsto_{\Gamma^*\mathit{CESK}_t^*\Xi} \hat{\varsigma}', \Xi'$ where $\hat{\varsigma}' = \varsigma'\{\kappa := \hat{\kappa}', \sigma := \sigma'\}$ and $\varsigma'.\kappa \in \mathit{unroll}_{\Xi'}(\hat{\kappa}')$ and finally there is an $L \in \mathit{live}^*(\hat{\varsigma}', \Xi')$ such that $\sigma'|_L = \varsigma'.\sigma|_{\mathit{live}(\varsigma')}$*
- **Completeness:** *if $\hat{\varsigma} \equiv \langle e, \rho, \sigma, \hat{\kappa}\rangle, \Xi \longmapsto_{\Gamma^*\mathit{CESK}_t^*\Xi} \hat{\varsigma}', \Xi'$ and there is an $L_\kappa \in K\mathcal{LL}(\Xi, \hat{\kappa})$ such that $\sigma|_L = \sigma$ (where $L = \mathcal{R}(\mathcal{T}(e, \rho) \cup L_\kappa, \sigma)$) and $\mathit{inv}(\hat{\varsigma}, \Xi)$, for all $\kappa \in \mathit{unroll}_\Xi(\hat{\kappa})$ such that $K\mathcal{LL}(\kappa) = L_\kappa$, there is a κ' such that $\hat{\varsigma}\{\hat{\kappa} := \kappa\} \longmapsto_{\Gamma\mathit{CESK}_t} \hat{\varsigma}'\{\hat{\kappa} := \kappa'\}$ (a GC step) and $\kappa' \in \mathit{unroll}_\Xi(\hat{\varsigma}'.\hat{\kappa})$*

Theorem 8 (Soundness of $\hat{\Gamma}\mathit{CESK}_t^*\,\Xi$)**.** *For all expressions e_0, if $\varsigma \longmapsto_{\Gamma\mathit{CESK}_t} \varsigma'$, $\mathit{inv}(\varsigma\{\kappa := \hat{\kappa}\}, \Xi)$, and $\varsigma.\kappa \in \mathit{unroll}_\Xi(\hat{\kappa})$, then there are $\Xi', \hat{\kappa}', \sigma''$ such that $\varsigma\{\kappa := \hat{\kappa}\}, \Xi \longmapsto_{\hat{\Gamma}\mathit{CESK}_t^*\Xi} \hat{\varsigma}', \Xi'$ where $\hat{\varsigma}' = \varsigma'\{\kappa := \hat{\kappa}', \sigma := \sigma''\}$ and $\varsigma'.\kappa \in \mathit{unroll}_{\Xi'}(\hat{\kappa}')$ and finally $\varsigma'.\sigma|_{\mathit{live}(\varsigma')} \sqsubseteq \sigma''|_{\widehat{\mathit{live}}(\hat{\varsigma}', \Xi')}$*

The proofs are straightforward, and use the usual lemmas for GC, such as idempotence of Γ and $\mathcal{T} \subseteq \mathcal{R}$.

4.2 Case study analyzing security features: the CM machine

The CM machine provides a model of access control: a dynamic branch of execution is given permission to use some resource if a continuation mark for that permission is set to "grant." There are three new forms we add to the lambda calculus to model this feature: `grant`, `frame`, and `test`. The `grant` construct adds a permission to the stack. The concern of unforgeable permissions is orthogonal, so we simplify with a set of permissions that is textually present in the program:

$P \in \mathit{Permissions}$ a set

$$\mathit{Expr} ::= \ldots \mid \texttt{grant}\ P\ e \mid \texttt{frame}\ P\ e \mid \texttt{test}\ P\ e\ e$$

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:

$$\mathcal{OK}(\varnothing, \kappa) = \mathit{True}$$

$$\mathcal{OK}(P, \epsilon^m) = \mathit{pass?}(P, m)$$

$$\mathcal{OK}(P, \phi{:}^m\kappa) = \mathit{pass?}(P, m) \textbf{ and } \mathcal{OK}(P \setminus m^{-1}(\mathbf{Grant}), \kappa)$$

where $\mathit{pass?}(P, m) = P \cap m^{-1}(\mathbf{Deny}) \overset{?}{=} \varnothing$

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

[6] The garbage collecting version of PDCFA [17] evaluates the $\hat{\Gamma}$ strategy.

[7] It is possible and more efficient to build the stack's live addresses piecemeal as an additional component of each state, precluding the need for $K\mathcal{LL}$. Each stack in Ξ would also store the live addresses to restore on pop.

$$\begin{array}{r|l}
\langle \texttt{grant}\ P\ e, \rho, \sigma, \kappa\rangle & \langle e, \rho, \sigma, \kappa[P \mapsto \mathbf{Grant}]\rangle \\
\langle \texttt{frame}\ P\ e, \rho, \sigma, \kappa\rangle & \langle e, \rho, \sigma, \kappa[\overline{P} \mapsto \mathbf{Deny}]\rangle \\
\langle \texttt{test}\ P\ e_0\ e_1, \rho, \sigma, \kappa\rangle & \langle e_0, \rho, \sigma, \kappa\rangle \text{ if } \mathit{True} = \mathcal{OK}(P, \kappa) \\
 & \langle e_1, \rho, \sigma, \kappa\rangle \text{ if } \mathit{False} = \mathcal{OK}(P, \kappa)
\end{array}$$

Figure 8. CM machine semantics

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.

$$m \in \mathit{PermissionMap} = \mathit{Permissions} \rightharpoonup \mathit{GD}$$
$$gd \in \mathit{GD} ::= \mathbf{Grant} \mid \mathbf{Deny}$$
$$\kappa \in \mathit{Kont} ::= \epsilon^m \mid \phi{:}^m \kappa$$

Update for continuation marks:

$$m[P \mapsto gd] = \lambda x.x \overset{?}{\in} P \to gd, m(x)$$
$$m[\overline{P} \mapsto gd] = \lambda x.x \overset{?}{\in} P \to m(x), gd$$

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

$$\begin{aligned}
\langle \texttt{test}\ P\ e_0\ e_1, \rho, \sigma, \hat{\kappa}\rangle, \Xi &\longmapsto \langle e_0, \rho, \sigma, \hat{\kappa}\rangle, \Xi \text{ if } \mathit{True} \in \widehat{\mathcal{OK}}(\Xi, P, \hat{\kappa}) \\
&\longmapsto \langle e_1, \rho, \sigma, \hat{\kappa}\rangle, \Xi \text{ if } \mathit{False} \in \widehat{\mathcal{OK}}(\Xi, P, \hat{\kappa})
\end{aligned}$$

where the a new $\widehat{\mathcal{OK}}$ function uses *terminal* and rewrite rules:

$$\begin{aligned}
\widehat{\mathcal{OK}}(\Xi, P, \hat{\kappa}) &= \mathit{terminal}(\longmapsto, \widehat{\mathcal{OK}}^*(\Xi, P, \hat{\kappa})) \\
\widehat{\mathcal{OK}}^*(\Xi, \varnothing, \hat{\kappa}) &\longmapsto \mathit{True} \\
\widehat{\mathcal{OK}}^*(\Xi, P, \epsilon^m) &\longmapsto \mathit{pass?}(P, m) \\
\widehat{\mathcal{OK}}^*(\Xi, P, \phi{:}^m \tau) &\longmapsto b \text{ where} \\
& \qquad b \in \{\mathit{pass?}(P, m) \textbf{ and } b \ :\ \hat{\kappa} \in \Xi(\tau), \\
& \qquad\quad b \in \widehat{\mathcal{OK}}(\Xi, P \setminus m^{-1}(\mathbf{Grant}), \hat{\kappa}))\}
\end{aligned}$$

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 $\widehat{OK}^*$.

We use the accumulator version in the proof for simplicity.

Lemma 9 (Correctness of $\widehat{\mathcal{OK}}$)**.** *For all $\Xi, P, \kappa, \hat{\kappa}$,*

- ***Soundness:*** *if $\kappa \in \mathit{unroll}_\Xi(\hat{\kappa})$ then $\mathcal{OK}(P, \kappa) \in \widehat{\mathcal{OK}}(\Xi, P, \hat{\kappa})$.*
- ***Completeness:*** *if $b \in \widehat{\mathcal{OK}}(\Xi, P, \hat{\kappa})$ then there is a $\kappa \in \mathit{unroll}_\Xi(\hat{\kappa})$ such that $b = \mathcal{OK}(P, \kappa)$.*

With this lemma in hand, the correctness proof is almost identical to the core proof of correctness.

Theorem 10 (Correctness)**.** *The abstract semantics is sound and complete in the same sense as Theorem 3.*

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. But, it was precisely the *irrelevance* that allowed the separation of σ and Ξ. 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.

$$\begin{array}{r|l}
\multicolumn{2}{c}{\varsigma \longmapsto_{SR} \varsigma'} \\
\hline
\mathbf{ev}(\texttt{reset}\ e, \rho, \sigma, \kappa, C) & \mathbf{ev}(e, \rho, \sigma, \epsilon, \kappa \circ C) \\
\mathbf{co}(\epsilon, \kappa \circ C, v, \sigma) & \mathbf{co}(\kappa, C, v, \sigma) \\
\mathbf{ev}(\texttt{shift}\ x.e, \rho, \sigma, \kappa, C) & \mathbf{ev}(e, \rho[x \mapsto a], \sigma', \epsilon, C) \\
\text{where} & \sigma' = \sigma \sqcup [a \mapsto \mathbf{comp}(\kappa)] \\
\mathbf{co}(\mathbf{fn}(\mathbf{comp}(\kappa')){:}\kappa, C, v, \sigma) & \mathbf{co}(\kappa', \kappa \circ C, v, \sigma)
\end{array}$$

Figure 9. Machine semantics for shift/reset

We introduce a new environment, χ, that maps these addresses to the stores they represent. The stores that contain addresses in χ are then *open*, and must be paired with χ 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, 27], providing processes isolation in operating systems [21], representing computational effects [13], modularly implementing error-correcting parsers [32], and finally undelimited 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 "**ev**al, **co**ntinue" 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, $\mathbf{comp}(\kappa)$, and a *meta-continuation*, $C \in \mathit{MKont} = \mathit{Kont}^*$ 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 stored 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 $\widehat{Kont}$ 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 χ. Contexts for meta-continuations are in two places: manual prompt introduction via `reset`, or via continuation invocation. At continuation capture time, continuation contexts are approximated to remove $\hat{\sigma}$ and χ components. The different context spaces are thus:

$$\begin{aligned}
\dot{\tau} \in \mathit{ExactContext} &::= \langle e, \rho, \hat{\sigma}, \chi\rangle \\
\hat{\tau} \in \widehat{\mathit{Context}} &::= \langle e, \rho, a\rangle \\
\tau \in \mathit{Context} &::= \hat{\tau} \mid \dot{\tau} \\
\gamma \in \mathit{MContext} &::= \langle e, \rho, \hat{\sigma}, \chi\rangle \mid \langle \tilde{\kappa}, \hat{v}, \hat{\sigma}, \chi\rangle
\end{aligned}$$

The approximation and flattening happens in $\mathbb{A}$:

$$\mathbb{A} : \mathit{KClosure} \times \mathit{Addr} \times \widetilde{\mathit{Kont}} \to \mathit{KClosure} \times \widetilde{\mathit{Kont}}$$

$$\begin{aligned}
\hat{\varsigma} \in \widehat{SR} &::= \mathbf{ev}(e, \rho, \hat{\sigma}, \chi, \hat{\kappa}, \hat{C}) \mid \mathbf{co}(\hat{\kappa}, \hat{C}, \hat{v}, \hat{\sigma}, \chi) \\
\mathit{State} &::= \hat{\varsigma}, \Xi_{\hat{\kappa}}, \Xi_{\hat{C}} \\
\chi \in \mathit{KClosure} &= \mathit{Addr} \rightharpoonup \wp(\mathit{Store}) \\
\Xi_{\hat{\kappa}} \in \mathit{KStore} &= \mathit{ExactContext} \rightharpoonup \wp(\widehat{\mathit{Kont}}) \\
\Xi_{\hat{C}} \in \mathit{CStore} &= \mathit{MContext} \rightharpoonup \wp(\widehat{\mathit{Kont}} \times \widehat{\mathit{MKont}}) \\
\hat{\kappa} \in \widehat{\mathit{Kont}} &::= \epsilon \mid \phi{:}\tau \mid \tau \qquad \tilde{\kappa} \in \widetilde{\mathit{Kont}} ::= \epsilon \mid \hat{\tau} \\
\hat{C} \in \widehat{\mathit{MKont}} &::= \epsilon \mid \gamma \qquad\qquad\quad \hat{v} \in \widehat{\mathit{Value}} ::= \tilde{\kappa} \mid (\ell, \rho)
\end{aligned}$$

Figure 10. Shift/reset abstract semantic spaces

$$\begin{aligned}
\mathbb{A}(\chi, a, \epsilon) &= \chi, \epsilon \\
\mathbb{A}(\chi, a, \phi{:}\tau) &= \chi', \phi{:}\hat{\tau} \text{ where } (\chi', \hat{\tau}) = \mathbb{A}(\chi, a, \tau) \\
\mathbb{A}(\chi, a, \langle e, \rho, \hat{\sigma}, \chi' \rangle) &= \chi \sqcup \chi' \sqcup [a \mapsto \hat{\sigma}], \phi{:}\langle e, \rho, a \rangle \\
\mathbb{A}(\chi, a, \langle e, \rho, b \rangle) &= \chi \sqcup [a \mapsto \chi(b)], \phi{:}\langle e, \rho, a \rangle
\end{aligned}$$

The third case is where continuation closures get flattened together. The fourth case is when an already approximate continuation is approximated: the approximation is inherited. Approximating the context and allocating the continuation in the store require two addresses, so we relax the specification of *alloc* to allow multiple address allocations in this case.

Each of the four rules of the original shift/reset machine has a corresponding rule that we explain piecemeal. We will use $\dashrightarrow$ for steps that do not modify the continuation stores for notational brevity. We use the above $\mathbb{A}$ function in the rule for continuation capture, as modified here.

$$\mathbf{ev}(\mathtt{shift}\ x.e, \rho, \hat{\sigma}, \chi, \hat{\kappa}, \hat{C}) \dashrightarrow \mathbf{ev}(e, \rho', \hat{\sigma}', \chi', \epsilon, \hat{C})$$

where

$$\begin{aligned}
(a, a') &= \mathit{alloc}(\hat{\varsigma}, \Xi_{\hat{\kappa}}, \Xi_{\hat{C}}) & \rho' &= \rho[x \mapsto a] \\
(\chi', \tilde{\kappa}) &= \mathbb{A}(\chi, a', \hat{\kappa}) & \hat{\sigma}' &= \hat{\sigma} \sqcup [a \mapsto \tilde{\kappa}]
\end{aligned}$$

The rule for `reset` stores the continuation and meta-continuation in $\Xi_{\hat{C}}$:

$$\begin{aligned}
\mathbf{ev}(\mathtt{reset}\ e, \rho, \hat{\sigma}, \chi, \hat{\kappa}, \hat{C}), \Xi_{\hat{\kappa}}, \Xi_{\hat{C}} &\longmapsto \mathbf{ev}(e, \rho, \hat{\sigma}, \chi, \epsilon, \gamma), \Xi_{\hat{\kappa}}, \Xi'_{\hat{C}} \\
&\text{where } \gamma = \langle e, \rho, \hat{\sigma}, \chi \rangle \\
&\qquad \Xi_{\hat{C}} = \Xi_{\hat{C}} \sqcup [\gamma \mapsto (\hat{\kappa}, \hat{C})]
\end{aligned}$$

The prompt-popping rule simply dereferences $\Xi_{\hat{C}}$:

$$\mathbf{co}(\epsilon, \gamma, \hat{v}, \hat{\sigma}, \chi) \dashrightarrow \mathbf{co}(\hat{\kappa}, \hat{C}, \hat{v}, \hat{\sigma}, \chi) \text{ if } (\hat{\kappa}, \hat{C}) \in \Xi_{\hat{C}}(\gamma)$$

The continuation installation rule extends $\Xi_{\hat{C}}$ at the different context:

$$\begin{aligned}
\mathbf{co}(\hat{\kappa}, \hat{C}, \hat{v}, \hat{\sigma}, \chi), \Xi_{\hat{\kappa}}, \Xi_{\hat{C}} &\longmapsto \mathbf{co}(\tilde{\kappa}, \gamma, \hat{v}, \hat{\sigma}, \chi), \Xi_{\hat{\kappa}}, \Xi'_{\hat{C}} \\
&\text{if } (\mathbf{appR}(\tilde{\kappa}), \hat{\kappa}') \in \mathit{pop}(\Xi_{\hat{\kappa}}, \chi, \hat{\kappa}) \\
&\text{where } \gamma = \langle \tilde{\kappa}, \hat{v}, \hat{\sigma}, \chi \rangle \\
&\qquad \Xi_{\hat{C}} = \Xi_{\hat{C}} \sqcup [\gamma \mapsto (\hat{\kappa}', \hat{C})]
\end{aligned}$$

Again we have a metafunction *pop*, but this time to interpret approximated continuations:

$$\mathit{pop}(\Xi_{\hat{\kappa}}, \chi, \hat{\kappa}) = \mathit{pop}^*(\hat{\kappa}, \varnothing)$$

where

$$\begin{aligned}
\mathit{pop}^*(\epsilon, G) &= \varnothing \\
\mathit{pop}^*(\phi{:}\tau, G) &= \{(\phi, \tau)\} \\
\mathit{pop}^*(\tau, G) &= \bigcup_{\hat{\kappa} \in G'} (\mathit{pop}^*(\hat{\kappa}, G \cup G')) \\
\text{where } G' &= \bigcup_{\dot{\tau} \in I(\tau)} \Xi_{\hat{\kappa}}(\dot{\tau}) \setminus G \\
I(\dot{\tau}) &= \{\dot{\tau}\} \\
I(\langle e, \rho, a \rangle) &= \{\langle e, \rho, \hat{\sigma}, \chi' \rangle \in \mathrm{dom}(\Xi_{\hat{\kappa}}) \ : \ \hat{\sigma} \in \chi(a), \chi' \sqsubseteq \chi\}
\end{aligned}$$

Notice that since we flatten χs 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:

```
(let* ([id (λ (x) x)]
       [f (λ (y) (shift k (k (k y))))]
       [g (λ (z) (reset (id (f z))))])
  (≤ (g 0) (g 1)))

(let* ([id (λ (x k) (k x))]
       [f (λ (y j) (j (j y)))]
       [g (λ (z h)
            (h (f z (λ (fv)
                       (id fv (λ (i) i))))))])
  (g 0 (λ (g0v) (g 1 (λ (g1v) (≤ g0v g1v))))))
```

The $\mathit{CESK}^*_t\,\Xi$ machine with a monovariant allocation strategy will predict the CPS'd 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'd version returns true.

5.3 Correctness

We impose an order on values since stored continuations are more approximate in the analysis than in SR:

$$\frac{}{v \sqsubseteq_{\Xi,\chi} v} \qquad \frac{\kappa \sqsubseteq \mathit{unroll}_{\Xi,\chi}(\tilde{\kappa})}{\mathbf{comp}(\kappa) \sqsubseteq_{\Xi,\chi} \tilde{\kappa}} \qquad \frac{\forall v \in \sigma(a). \exists \hat{v} \in \hat{\sigma}(a). v \sqsubseteq_{\Xi,\chi} \hat{v}}{\sigma \sqsubseteq_{\Xi,\chi} \hat{\sigma}}$$

$$\frac{\begin{array}{c}\kappa \sqsubseteq \mathit{unroll}_{\Xi_{\hat{\kappa}},\chi}(\hat{\kappa}) \\ C \sqsubseteq \mathit{unrollC}_{\Xi_{\hat{\kappa}},\Xi_{\hat{C}},\chi}(\hat{C}) \qquad \sigma \sqsubseteq_{\Xi_{\hat{\kappa}},\chi} \hat{\sigma}\end{array}}{\mathbf{ev}(e, \rho, \sigma, \kappa, C) \sqsubseteq \mathbf{ev}(e, \rho, \hat{\sigma}, \chi, \hat{\kappa}, \hat{C}), \Xi_{\hat{\kappa}}, \Xi_{\hat{C}}}$$

$$\frac{\begin{array}{c}v \sqsubseteq_{\Xi_{\hat{\kappa}},\chi} \hat{v} \qquad \kappa \sqsubseteq \mathit{unroll}_{\Xi_{\hat{\kappa}},\chi}(\hat{\kappa}) \\ C \sqsubseteq \mathit{unrollC}_{\Xi_{\hat{\kappa}},\Xi_{\hat{C}},\chi}(\hat{C}) \qquad \sigma \sqsubseteq_{\Xi_{\hat{\kappa}},\chi} \hat{\sigma}\end{array}}{\mathbf{co}(\kappa, C, v, \sigma) \sqsubseteq \mathbf{co}(\hat{\kappa}, \hat{C}, \hat{v}, \hat{\sigma}, \chi), \Xi_{\hat{\kappa}}, \Xi_{\hat{C}}}$$

Unrolling differs from the previous sections because the values in frames can be approximate. Thus, instead of expecting the exact continuation to be in the unrolling, we have a judgment that an unrolling approximates a given continuation in Figure 11 (note we reuse I from pop^*'s definition).

Theorem 11 (Soundness)**.** *If* $\lozenge \longmapsto_{SR} \blacklozenge$*, and* $\lozenge \sqsubseteq \square$ *then there is* $\blacksquare$ *such that* $\square \longmapsto_{SRS\chi_t} \blacksquare$ *and* $\blacklozenge \sqsubseteq \blacksquare$*.*

Freshness implies completeness The high level proof idea is that fresh allocation separates evaluation into a sequence of bounded length paths that have the same store, but the store only grows and distinguishes contexts such that each continuation and metacontinuation have a unique unrolling. It is an open question whether the addition of garbage collection preserves completeness. Each context with the same store will have different expressions in them since expressions can only get smaller until a function call, at which point the store grows. This forms an order on contexts: smaller store means smaller context, and same store but smaller expression (indeed a subexpression) means a smaller context. Every entry in each enviroment $(\hat{\sigma}, \chi, \Xi_{\hat{\kappa}}, \Xi_{\hat{C}})$ will map to a unique element, and the continuation stores will have no circular references (the context in the tail of a continuation is strictly smaller than the context that maps to the continuation). There can only be one context that I maps to for approximate contexts because of the property of stores in contexts.

$$\frac{}{\mathbf{appL}(e,\rho) \sqsubseteq_{\Xi,\chi} \mathbf{appL}(e,\rho)} \qquad \frac{v \sqsubseteq_{\Xi,\chi} \hat{v}}{\mathbf{appR}(v) \sqsubseteq_{\Xi,\chi} \mathbf{appR}(\hat{v})}$$

$$\frac{}{\epsilon \sqsubseteq \mathit{unroll}_{\Xi,\chi}(\epsilon)} \qquad \frac{\phi \sqsubseteq_{\Xi,\chi} \hat{\phi} \qquad \kappa \sqsubseteq \mathit{unroll}_{\Xi,\chi}(\tau)}{\phi{:}\kappa \sqsubseteq \mathit{unroll}_{\Xi,\chi}(\hat{\phi}{:}\tau)}$$

$$\frac{\hat{\kappa} \in \Xi(\dot{\tau}) \quad \kappa \sqsubseteq \mathit{unroll}_{\Xi,\chi}(\hat{\kappa})}{\kappa \sqsubseteq \mathit{unroll}_{\Xi,\chi}(\dot{\tau})} \qquad \frac{\dot{\tau} \in I(\Xi,\chi,\hat{\tau}) \quad \kappa \sqsubseteq \mathit{unroll}_{\Xi,\chi}(\dot{\tau})}{\kappa \sqsubseteq \mathit{unroll}_{\Xi,\chi}(\hat{\tau})}$$

$$\frac{}{\epsilon \sqsubseteq \mathit{unrollC}_{\Xi_{\hat{\kappa}},\Xi_{\hat{C}},\chi}(\epsilon)}$$

$$\frac{\begin{array}{c}(\hat{\kappa},\hat{C}) \in \Xi_{\hat{C}}(\gamma) \\ \kappa \sqsubseteq \mathit{unroll}_{\Xi_{\hat{\kappa}},\chi}(\hat{\kappa}) \qquad C \sqsubseteq \mathit{unrollC}_{\Xi_{\hat{\kappa}},\Xi_{\hat{C}},\chi}(\hat{C})\end{array}}{\kappa \circ C \sqsubseteq \mathit{unrollC}_{\Xi_{\hat{\kappa}},\Xi_{\hat{C}},\chi}(\gamma)}$$

Figure 11. Order on (meta-)continuations

We distill these intuitions into an invariant about states that we will then use to prove completeness.

$$\frac{\begin{array}{c}\forall a \in \mathrm{dom}(\hat{\sigma}).\exists \hat{v}.\hat{\sigma}(a) = \{\hat{v}\} \wedge \hat{v} \preceq_{\chi} \Xi_{\hat{\kappa}} \\ \forall a \in \mathrm{dom}(\chi).\exists \hat{\sigma}'.\chi(a) = \{\hat{\sigma}'\} \wedge \hat{\sigma}' \in \pi_3(\mathrm{dom}(\Xi_{\hat{\kappa}})) \\ \forall \dot{\tau} \in \mathrm{dom}(\Xi_{\hat{\kappa}}).\exists \hat{\kappa}.\Xi_{\hat{\kappa}}(\dot{\tau}) = \{\hat{\kappa}\} \wedge \hat{\kappa} \sqsubset_{\chi}^{\Xi_{\hat{\kappa}}} \dot{\tau} \\ \forall \gamma \in \mathrm{dom}(\Xi_{\hat{C}}).\exists \hat{C}.\Xi_{\hat{C}}(\gamma) = \{\hat{C}\} \wedge \hat{C} \sqsubset \gamma\end{array}}{\mathit{inv}^*(\hat{\sigma},\chi,\Xi_{\hat{\kappa}},\Xi_{\hat{C}})}$$

$$\frac{\begin{array}{c}\mathit{inv}^*(\hat{\sigma},\chi,\Xi_{\hat{\kappa}},\Xi_{\hat{C}}) \qquad \langle e,\rho,\hat{\sigma},\chi\rangle \sqsubset \mathrm{dom}(\Xi_{\hat{\kappa}}) \cup \mathrm{dom}(\Xi_{\hat{C}}) \\ (\exists \langle e_c,\rho,\hat{\sigma},\chi\rangle \in \mathrm{dom}(\Xi_{\hat{\kappa}})) \implies e \in \mathit{subexpressions}(e_c) \\ \hat{\kappa} \preceq_{\chi} \Xi_{\hat{\kappa}} \qquad \hat{C} \preceq \Xi_{\hat{C}}\end{array}}{\mathit{inv}_{\mathit{fresh}}(\mathbf{ev}(e,\rho,\hat{\sigma},\chi,\hat{\kappa},\hat{C}),\Xi_{\hat{\kappa}},\Xi_{\hat{C}})}$$

$$\frac{\mathit{inv}^*(\hat{\sigma},\chi,\Xi_{\hat{\kappa}},\Xi_{\hat{C}}) \qquad \hat{v} \preceq_{\chi} \Xi_{\hat{\kappa}} \qquad \hat{\kappa} \preceq_{\chi} \Xi_{\hat{\kappa}} \qquad \hat{C} \preceq \Xi_{\hat{C}}}{\mathit{inv}_{\mathit{fresh}}(\mathbf{co}(\hat{\kappa},\hat{C},\hat{v},\hat{\sigma},\chi),\Xi_{\hat{\kappa}},\Xi_{\hat{C}})}$$

Where the order ≤ states that any contexts in the (meta-)continuation are mapped in the given table.

$$\frac{}{(\ell,\rho) \preceq_{\chi} \Xi_{\hat{\kappa}}} \qquad \frac{}{\epsilon \preceq_{\chi} \Xi_{\hat{\kappa}}} \qquad \frac{}{\epsilon \preceq \Xi_{\hat{C}}} \qquad \frac{\dot{\tau} \in \mathrm{dom}(\Xi_{\hat{\kappa}})}{\dot{\tau} \preceq_{\chi} \Xi_{\hat{\kappa}}} \qquad \frac{\gamma \in \mathrm{dom}(\Xi_{\hat{C}})}{\gamma \preceq \Xi_{\hat{C}}}$$

$$\frac{\exists \hat{\sigma}.\chi(a) = \{\hat{\sigma}\} \qquad \exists! \chi'.\langle e,\rho,\hat{\sigma},\chi'\rangle \in \mathrm{dom}(\Xi_{\hat{\kappa}}) \wedge \chi' \sqsubseteq \chi}{\langle e,\rho,a\rangle \preceq_{\chi} \Xi_{\hat{\kappa}}}$$

And the order ⊏ states that the contexts in the (meta-)continuation are strictly smaller than the given context.

$$\frac{}{\epsilon \sqsubset_{\chi}^{\Xi_{\hat{\kappa}}}} \qquad \frac{}{\epsilon \sqsubset \gamma} \qquad \frac{\tau \sqsubset_{\chi}^{\Xi_{\hat{\kappa}}} \dot{\tau}}{\phi{:}\tau \sqsubset_{\chi}^{\Xi_{\hat{\kappa}}} \dot{\tau}}$$

$$\frac{e' \in \mathit{subexpressions}(e)}{\langle e',\rho,\hat{\sigma},\chi\rangle \sqsubset_{\chi}^{\Xi_{\hat{\kappa}}} \langle e,\rho,\hat{\sigma},\chi\rangle} \qquad \frac{e' \in \mathit{subexpressions}(e)}{\langle e',\rho,\hat{\sigma},\chi\rangle \sqsubset \langle e,\rho,\hat{\sigma},\chi\rangle}$$

$$\frac{\mathrm{dom}(\hat{\sigma}) \sqsubset \mathrm{dom}(\hat{\sigma}')}{\langle \_,\_,\hat{\sigma},\_\rangle \sqsubset_{\chi}^{\Xi_{\hat{\kappa}}} \langle \_,\_,\hat{\sigma}',\_\rangle} \qquad \frac{\mathrm{dom}(\hat{\sigma}) \sqsubset \mathrm{dom}(\hat{\sigma}')}{\langle \_,\_,\hat{\sigma},\_\rangle \sqsubset \langle \_,\_,\hat{\sigma}',\_\rangle}$$

$$\frac{\forall \dot{\tau}' \in I(\Xi_{\hat{\kappa}},\chi,\hat{\tau}) \qquad \dot{\tau}' \sqsubset_{\chi}^{\Xi_{\kappa}} \dot{\tau}}{\hat{\tau} \sqsubset_{\chi}^{\Xi_{\hat{\kappa}}} \dot{\tau}}$$

Lemma 12 (Freshness invariant)**.** *If alloc produces fresh addresses, $\mathit{inv}_{\mathit{fresh}}(\hat{\varsigma},\Xi_{\hat{\kappa}},\Xi_{\hat{C}})$ and $\hat{\varsigma},\Xi_{\hat{\kappa}},\Xi_{\hat{C}} \longmapsto \hat{\varsigma}',\Xi'_{\hat{\kappa}},\Xi'_{\hat{C}}$ then $\mathit{inv}_{\mathit{fresh}}(\hat{\varsigma}',\Xi'_{\hat{\kappa}},\Xi'_{\hat{C}})$.*

Proof. By case analysis on the step. □

Theorem 13 (Complete for fresh allocation)**.** *If alloc produces fresh addresses then the resulting semantics is complete with respect to states satisfying the invariant.*

Proof sketch. By case analysis and use of the invariant to exploit the fact the unrollings are unique and the singleton codomains pigeon-hole the possible steps to only concrete ones. □

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

$$\begin{aligned} M \in \mathit{Memo} &= \mathit{Context} \rightharpoonup \wp(\mathit{Relevant}) \\ \mathit{Relevant} &::= \langle e,\rho,\sigma\rangle \end{aligned}$$

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 3.2 with global Ξ and M, taking care to
   (a) at memo-use time, still extend Ξ so later memo table extensions will "flow" to previous memo table uses, and
   (b) when Ξ 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 Ξ.

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

$$\begin{array}{r|l}
\multicolumn{2}{c}{\hat{\varsigma}, \Xi, M \longmapsto \hat{\varsigma}', \Xi', M'} \\
\hline
\langle (e_0\ e_1), \rho, \sigma, \hat{\kappa} \rangle, \Xi, M & \langle e_0, \rho, \sigma, \mathbf{appL}(e_1, \rho){:}\tau \rangle, \Xi, M \\
 & \quad \text{if } \tau \notin \mathrm{dom}(M), \text{ or} \\
 & \langle e', \rho', \sigma', \hat{\kappa} \rangle, \Xi', M \\
 & \quad \text{if } \langle e', \rho', \sigma' \rangle \in M(\tau) \\
\text{where} & \tau = \langle (e_0\ e_1), \rho, \sigma \rangle \\
 & \Xi' = \Xi \sqcup [\tau \mapsto \hat{\kappa}] \\
\langle v, \sigma, \mathbf{appR}(\lambda x.e, \rho){:}\tau \rangle, \Xi, M & \langle e, \rho', \sigma', \hat{\kappa} \rangle, \Xi, M' \text{ if } \hat{\kappa} \in \Xi(\tau) \\
\text{where} & \rho' = \rho[x \mapsto a] \\
 & \sigma' = \sigma \sqcup [a \mapsto v] \\
 & M' = M \sqcup [\tau \mapsto \langle e, \rho', \sigma' \rangle]
\end{array}$$

Figure 12. Important memoization rules

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 semanticists' 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.

Context-irrelevance is a property of the semantics *without* continuation stores, so there is an additional invariant to that of section 3 for the semantics with Ξ and M: M respects context irrelevance.

$$\frac{\begin{array}{c}\mathrm{dom}(M) \subseteq \mathrm{dom}(\Xi) \\ \forall \tau \equiv \langle e_c, \rho_c, \sigma_c \rangle \in \mathrm{dom}(M), \langle e_r, \rho_r, \sigma_r \rangle \in M(\tau), \\ \hat{\kappa} \in \Xi(\tau), \kappa \in \mathit{unroll}_{\Xi}(\hat{\kappa}). \\ \exists \pi \equiv \langle e_c, \rho_c, \sigma_c, \kappa \rangle \longmapsto^{*}_{CESK_t} \langle e_r, \rho_r, \sigma_r, \kappa \rangle . \mathit{ht}(\pi, \kappa)\end{array}}{\mathit{inv}_M(\Xi, M)}$$

Inexact GC does *not* respect context irrelevance for the same reasons it is not complete: some states are spurious, meaning some memo table entries will be spurious, and the expected path in the invariant will not exist. The reason we use unrolled continuations instead of simply ϵ for this (balanced) path is precisely for stack inspection reasons.

The rules in Figure 12 are the importantly changed rules from section 3 that short-circuit to memoized results. The technique looks more like memoization with a $CESIK_t^{*}\Xi$ 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*.

$$\mathcal{F}_e(S, R, F, \Xi, M) = (S \cup F, R \cup R', F' \setminus S, \Xi', M')$$

where

$$\begin{aligned}
I &= \bigcup_{\varsigma \in F} \{ (\langle \varsigma, \varsigma' \rangle, \Xi', M') \ : \ \varsigma, \Xi, M \longmapsto \varsigma', \Xi', M' \} \\
R' &= \pi_0 I \qquad\qquad \Xi' = \bigsqcup \pi_1 I \qquad\qquad M' = \bigsqcup \pi_2 I \\
\Delta\Xi &= \Xi' \setminus \Xi \qquad \Delta M = M' \setminus M \\
F' &= \pi_1 R' \cup \{ \langle e, \rho, \sigma, \hat{\kappa} \rangle : \tau \in \mathrm{dom}(\Delta\Xi) \cap \mathrm{dom}(\Delta M). \\
&\qquad \hat{\kappa} \in \Delta\Xi(\tau), \langle e, \rho, \sigma \rangle \in \Delta M(\tau) \}
\end{aligned}$$

Theorem 14 (Correctness)**.** *Same property is the same as in Theorem 4, where reify ignores the M component.*

The proof appeals to the invariant on M whose proof involves an additional argument for the short-circuiting step that reconstructs the path from a memoized result using both context irrelevance and the table invariants.

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 `call/cc` by removing 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 $\hat{\Gamma}$).

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 store component of states when there is a global abstract store, an assumption that does not hold for garbage collection. The implementation[8] 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 [18] 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 $\mathcal{O}(n^3)$ 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 Ξ[9]. 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 [35].

[8] `http://github.com/dvanhorn/oaam`

[9] See `gluck.rkt` in online materials for Glück's style

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 [3], 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 [31].

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

[1] Massimo Bartoletti, Pierpaolo Degano, and Gian L. Ferrari. Stack inspection and secure program transformations. *International Journal of Information Security*, 2(3-4):187–217, 2004.

[2] 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.

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

[4] Byeong-Mo Chang. Static check analysis for java stack inspection. *SIGPLAN Notices*, 41(3):40–48, 2006.

[5] John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Trans. Program. Lang. Syst.*, 26(6):1029–1052, 2004.

[6] Stephen A. Cook. Linear time simulation of deterministic two-way pushdown automata. In *IFIP Congress (1)*, pages 75–80, 1971.

[7] P Cousot, R Cousot, J Feret, L Mauborgne, A Miné, D Monniaux, and X Rival. Varieties of static analyzers: A comparison with textscAstrée, invited paper. In He Jifeng and J Sanders, editors, *Proc. First IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE$*
,$'07, pages 3–17. IEEE Computer Society Press, Los Alamitos, California, United States, 2007.

[8] Olivier Danvy. Defunctionalized interpreters for programming languages. In James Hook and Peter Thiemann, editors, *ICFP*, pages 131–142. ACM, 2008.

[9] 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.

[10] Christopher Earl, Matthew Might, and David Van Horn. Pushdown Control-Flow analysis of Higher-Order programs. In *Workshop on Scheme and Functional Programming*, 2010.

[11] 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.

[12] Matthias Felleisen, Robert B. Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[13] Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 446–457. ACM, 1994.

[14] Robert Glück. Simulation of two-way pushdown automata revisited. In Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh, and John Hatcliff, editors, *Festschrift for Dave Schmidt*, volume 129 of *EPTCS*, pages 250–258, 2013.

[15] Ben Hardekopf, Ben Wiedermann, Berkeley R. Churchill, and Vineeth Kashyap. Widening for Control-Flow. In Kenneth L. McMillan and Xavier Rival, editors, *VMCAI*, volume 8318 of *Lecture Notes in Computer Science*, pages 472–491. Springer, 2014.

[16] J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. Optimizing abstract abstract machines. In Greg Morrisett and Tarmo Uustalu, editors, *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. ACM SIGPLAN, ACM Press, 2013.

[17] J. Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. Pushdown flow analysis with abstract garbage collection. *Journal of Functional Programming*, 24(2-3):218–283, 2014.

[18] N. D. Jones. A note on linear time simulation of deterministic two-way pushdown automata. 6(4):110–112, August 1977. ISSN 0020-0190 (print), 1872-6119 (electronic).

[19] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: Designing a sound, configurable, and efficient static analyzer for JavaScript. *CoRR*, abs/1403.3996, 2014.

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

[21] Oleg Kiselyov and Chung-Chieh Shan. Delimited continuations in operating systems. pages 291–302. 2007.

[22] Jay McCarthy. Concerning PLT webserver commit 72ec6342ea. private communication.

[23] Matthew Might and Panagiotis Manolios. A posteriori soundness for non-deterministic abstract interpretations. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 260–274. Springer-Verlag, 2009.

[24] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM International Conference on Functional Programming (ICFP 2006)*, pages 13–25, 2006.

[25] 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.

[26] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *PLDI*, pages 229–238. ACM, 2012.

[27] Christian Queinnec. Continuations and web servers. *Higher-Order and Symbolic Computation*, 17(4):277–295, 2004.

[28] John H. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 104–118. ACM, 1977.

[29] 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.

[30] 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.

[31] Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190. Springer, 2005. ISBN 3-540-25333-5.

[32] Olin Shivers and Aaron J. Turon. Modular rollback through control logging: a pair of twin functional pearls. In *ICFP*, pages 58–68, 2011.

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

[34] David Van Horn and Matthew Might. Systematic abstraction of abstract machines. *Journal of Functional Programming*, 22(Special Issue 4-5):705–746, 2012.

[35] Dimitrios Vardoulakis. *CFA2: Pushdown Flow Analysis for Higher-Order Languages*. PhD thesis, Northeastern University, 2012.

[36] 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, 2011.

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

[38] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.

[39] Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. An empirical study on classification methods for alarms from a bug-finding static C analyzer. *Inf. Process. Lett.*, 102(2-3):118–123, 2007.