Abstracting Abstract Machines

A Systematic Approach to Higher-Order Program Analysis

David Van Horn*
Northeastern University
Boston, Massachusetts
dvanhorn@ccs.neu.edu

Matthew Might
University of Utah
Salt Lake City, Utah
might@cs.utah.edu

ABSTRACT

Predictive models are fundamental to engineering reliable software systems. However, designing conservative, computable approximations for the behavior of programs (static analyses) remains a difficult and error-prone process for modern high-level programming languages. What analysis designers need is a principled method for navigating the gap between semantics and analytic models: analysis designers need a method that tames the *interaction* of complex languages features such as higher-order functions, recursion, exceptions, continuations, objects and dynamic allocation.

We contribute a *systematic approach to program analysis* that yields novel and transparently sound static analyses. Our approach relies on existing derivational techniques to transform high-level language semantics into low-level deterministic state-transition systems (with potentially infinite state spaces). We then perform a series of simple machine refactorings to obtain a sound, computable approximation, which takes the form of a *non-deterministic* state-transition systems with *finite* state spaces. The approach scales up uniformly to enable program analysis of realistic language features, including higher-order functions, tail calls, conditionals, side effects, exceptions, first-class continuations, and even garbage collection.

1. INTRODUCTION

Software engineering, compiler optimizations, program parallelization, system verification, and security assurance depend on program analysis, a ubiquitous and central theme of programming language research. At the same time, the production of modern software systems employs expressive, higher-order languages such as Java, JavaScript, C#, Python, Ruby, etc., implying a growing need for fast, precise, and scalable higher-order program analyses.

The original version of this paper was published as "Abstracting Abstract Machines" in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming.*

*Supported by the National Science Foundation under grant 0937060 to the Computing Research Association for the CIFellow Project.

Program analysis aims to soundly predict properties of programs before being run. (*Sound* in program analysis means "conservative approximation": if a sound analysis says a program must not exhibit behavior, then that program will not exhibit that behavior; but if a sound analysis says a program may exhibit a behavior, then it may or may not exhibit that behavior.) For over thirty years, the research community has expended significant effort designing effective analyses for higher-order programs [13]. Past approaches have focused on connecting high-level language semantics such as structured operational semantics, denotational semantics, or reduction semantics to equally high-level but dissimilar analytic models. These models are too often far removed from their programming language counterparts and take the form of constraint languages specified as relations on sets of program fragments [25, 18, 12]. These approaches require significant ingenuity in their design and involve complex constructions and correctness arguments, making it difficult to establish soundness, design algorithms, or grow the language under analysis. Moreover, such analytic models, which focus on "value flow", *i.e.*, determining which syntactic values may show up at which program sites at run-time, have a limited capacity to reason about many low-level intensional properties such as memory management, stack behavior, or trace-based properties of computation. Consequently, higher-order program analysis has had limited impact on large-scale systems, despite the apparent potential for program analysis to aid in the construction of reliable and efficient software.

In this paper, we describe a *systematic approach to program analysis* that overcomes many of these limitations by providing a straightforward derivation process, lowering verification costs and accommodating sophisticated language features and program properties.

Our approach relies on leveraging existing techniques to transform high-level language semantics into *abstract machines*—low-level deterministic state-transition systems with potentially infinite state spaces. Abstract machines [11], and the paths from semantics to machines [20, 5, 7], have a long history in the research on programming languages.

From an abstract machine, which represents the idealized core of a realistic run-time system, we perform a series of basic machine refactorings to obtain a *non-deterministic* state-transition system with a *finite* state space. The refactorings are simple: (1) variable bindings and the control stack are redirected through the machine's store and (2) the store is bounded to a finite size. Due to finiteness, store updates must become merges, leading to the possibility of multiple values residing in a single store location. This in turn requires store look-ups be replaced by a non-deterministic choice among the multiple values at a given location. The derived machine computes a sound approximation of the original machine, and thus forms an *abstract interpretation* of the machine and the high-level semantics.

The approach scales up uniformly to enable program analysis of realistic language features, including higher-order functions, tail calls, conditionals, side effects, exceptions, first-class continuations, and even garbage collection. Thus, we are able to refashion semantic techniques used to model language features into abstract interpretation techniques for reasoning about the behavior of those very same features.

Background and notation: We present a brief introduction to reduction semantics and abstract machines. For background and a more extensive introduction to the concepts, terminology, and notation employed in this paper, we refer the reader to *Semantics Engineering with PLT Redex* [7].

2. FROM SEMANTICS TO MACHINES AND MACHINES TO ANALYSES

In this section, we demonstrate our systematic approach to analysis by stepping through a derivation from the high-level semantics of a prototypical higher-order programming language to a low-level abstract machine, and from the abstract machine to a sound and computable analytic model that predicts intensional properties of that machine. As a prototypical language, we choose the call-by-value λ-calculus [19], a core computational model for both functional and object-oriented languages. We choose to model program behavior with a simple operational model given in the form of a reduction semantics. Despite this simplicity, reduction semantics scale to full-fledged programming languages [22], although the choice is somewhat arbitrary since it is known how to construct abstract machines from a number of semantic paradigms [5]. In subsequent sections, we demonstrate the approach handles richer language features such as control, state, and garbage collection, and we have successfully employed the same method to statically reason about language features such as laziness, exceptions, and stack-inspection, and programming languages such as Java and JavaScript. In all cases, analyses are derived following the systematic approach presented here.

2.1 Reduction semantics

To begin, consider the following language of expressions:

$$\begin{array}{ll} e \in \mathit{Exp} = x \mid (ee) \mid (\lambda x.e) & \\ x \in \mathit{Var} & \text{an infinite set of identifiers.} \end{array}$$

The syntax of expressions includes variables, applications, and functions. Values v, for the purposes of this language, include only function terms, $(\lambda x.e)$. We say x is the *formal parameter* of the function $(\lambda x.e)$, and e is its *body*. A *program* is a closed expression, *i.e.*, an expression in which every variable occurs within some function that binds that variable as its formal parameter. Call-by-value *reduction* is characterized by the relation $\mathbf{v}$:

$$((\lambda x.e)v) \quad \mathbf{v} \quad [v/x]e,$$

which states that a function applied to a value reduces to the body of the function with every occurrence of the formal parameter replaced by the value. The expression on the left-hand side is a known as a *redex* and the right-hand side is its *contractum*.

Reduction can occur within a context of an *evaluation context*, defined by the following grammar:

$$E = [\,] \mid (Ee) \mid (vE).$$

An evaluation context can be thought of as an expression with a single "hole" in it, which is where a redex may be reduced. It is straightforward to observe that for all programs, either the program is a value, or it decomposes uniquely into an evaluation context and redex, written $E[((\lambda x.e)v)]$. Thus the grammar as given specifies a deterministic reduction strategy, which is formalized as a *standard reduction relation* on programs:

$$E[e] \longmapsto_{\mathbf{v}} E[e'], \text{ if } e \;\mathbf{v}\; e'.$$

The *evaluation* of a program is defined by a partial function relating programs to values [7, page 67]:

$$\mathit{eval}(e) = v \text{ if } e \longmapsto\!\!\!\!\!\rightarrow_{\mathbf{v}} v, \text{ for some } v,$$

where $\longmapsto\!\!\!\!\!\rightarrow_{\mathbf{v}}$ denotes the reflexive, transitive closure of the standard reduction relation.

We have now established the high-level semantic basis for our prototypical language. The semantics is in the form of an evaluation function defined by the reflexive, transitive closure of the standard reduction relation. However, the evaluation function as given does not shed much light on a realistic implementation. At each step, the program is traversed according to the grammar of evaluation contexts until a redex is found. When found, the redex is reduced and the contractum is plugged back into the context. The process is then repeated, again traversing from the beginning of the program. Abstract machines offer an extensionally equivalent but more realistic model of evaluation that short-cuts the plugging of a contractum back into a context and the subsequent decomposition [6].

2.2 CEK machine

The CEK machine [20, Interpreter III][7, page 100] is a state transition system that efficiently performs evaluation of a program. There are two key ideas in its construction, which can be carried out systematically [2]. The first is that substitution, which is not a viable implementation strategy, is instead represented in a delayed, explicit manner as an *environment* structure. So a substitution $[v/x]e$ is represented by e and an environment that maps x to v. Since e and v may have previous substitutions applied, this will likewise be represented with environments. So in general, if ρ is the environment of e and ρ' is the environment of v, then we represent $[v/x]e$ by e in the environment ρ extended with a mapping of x to (v, ρ'), written $\rho[x \mapsto (v, \rho')]$. The pairing of a value and an environment is known as a *closure* [11].

The second key idea is that evaluation contexts are constructed inside-out and represent continuations:

1. $[\,]$ is represented by $\mathbf{mt}$;
2. $E[([\,]e)]$ is represented by $\mathbf{ar}(e', \rho, \kappa)$ where ρ closes e' to represent e and κ represents E; and
3. $E[(v[\,])]$ is represented by $\mathbf{fn}(v', \rho, \kappa)$ where ρ closes v' to represent v and κ represents E.

In this way, evaluation contexts form a program stack: $\mathbf{mt}$ is the empty stack, and $\mathbf{ar}$ and $\mathbf{fn}$ are frames.

$$\varsigma \longmapsto_{CEK} \varsigma'$$

$\langle x, \rho, \kappa\rangle$	$\langle v, \rho', \kappa\rangle$ where $\rho(x) = (v, \rho')$
$\langle (e_0 e_1), \rho, \kappa\rangle$	$\langle e_0, \rho, \mathbf{ar}(e_1, \rho, \kappa)\rangle$
$\langle v, \rho, \mathbf{ar}(e, \rho', \kappa)\rangle$	$\langle e, \rho', \mathbf{fn}(v, \rho, \kappa)\rangle$
$\langle v, \rho, \mathbf{fn}((\lambda x.e), \rho', \kappa)\rangle$	$\langle e, \rho'[x \mapsto (v, \rho)], \kappa\rangle$

Figure 1: CEK machine.

States of the CEK machine are triples consisting of an expression, an environment that closes the control string, and a continuation:

$$\begin{aligned} \varsigma \in \Sigma &= \mathit{Exp} \times \mathit{Env} \times \mathit{Cont} \\ v \in \mathit{Val} &= (\lambda x.e) \\ \rho \in \mathit{Env} &= \mathit{Var} \rightarrow_{\text{fin}} \mathit{Val} \times \mathit{Env} \\ \kappa \in \mathit{Cont} &= \mathbf{mt} \mid \mathbf{ar}(e, \rho, \kappa) \mid \mathbf{fn}(v, \rho, \kappa). \end{aligned}$$

The transition function for the CEK machine is defined in Figure 1. The initial machine state for a program e is given by the inj_{CEK} function:

$$\mathit{inj}_{CEK}(e) = \langle e, \emptyset, \mathbf{mt}\rangle.$$

Evaluation is defined by the reflexive, transitive closure of the machine transition relation and a "*real*" function [19, page 129] that maps closures to the term represented:

$$\mathit{eval}_{CEK}(e) = \mathit{real}(v, \rho), \text{ where } \mathit{inj}_{CEK}(e) \longmapsto\!\!\!\!\to_{\mathbf{v}} \langle v, \rho, \mathbf{mt}\rangle,$$

which is equivalent to the *eval* function of Section 2.1:

Lemma 1 (CEK Correctness [7]) $\mathit{eval}_{CEK} = \mathit{eval}$.

We have now established a correct low-level evaluator for our prototypical language that is extensionally equivalent to the high-level reduction semantics. However, program analysis is not just concerned with the result of a computation, but also with *how* it was produced, *i.e.*, analysis should predict intensional properties of the machine as it runs a program. We therefore adopt a reachable states semantics that relates a program to the set of all its intermediate steps:

$$\mathit{CEK}(e) = \{\varsigma \mid \mathit{inj}_{CEK}(e) \longmapsto\!\!\!\!\to_{CEK} \varsigma\}.$$

Membership in the set of reachable states is straightforwardly undecidable. The goal of analysis, then, is to construct an *abstract interpretation* [4] that is a sound and computable approximation of the *CEK* function.

We can do this by constructing a machine that is similar in structure to the CEK machine: it is defined by an *abstract state transition* relation, $\longmapsto_{\widehat{CEK}}$, which operates over *abstract states*, $\hat{\Sigma}$, that approximate states of the CEK machine. Abstract evaluation is then defined as:

$$\widehat{\mathit{CEK}}(e) = \{\hat{\varsigma} \mid \mathit{inj}_{\widehat{CEK}}(e) \longmapsto\!\!\!\!\to_{\widehat{CEK}} \hat{\varsigma}\}.$$

1. *Soundness* is achieved by showing transitions preserves approximation, so that if $\varsigma \longmapsto_{CEK} \varsigma'$ and $\hat{\varsigma}$ approximates ς, then there exists an abstract state $\hat{\varsigma}'$ such that $\hat{\varsigma} \longmapsto_{\widehat{CEK}} \hat{\varsigma}'$ and $\hat{\varsigma}'$ approximates ς'.

2. *Decidability* is achieved by constructing the approximation in such a way that the state-space of the abstracted machine is finite, which guarantees that for any program e, the set $\widehat{\mathit{CEK}}(e)$ is finite.

An attempt at approximation: A simple approach to abstracting the machine's state space is to apply a *structural abstraction*, which lifts approximation across the structure of a machine state, *i.e.*, expressions, environments, and continuations. The problem with the structural abstraction approach for the CEK machine is that both environments and continuations are recursive structures. As a result, the abstraction yields objects in an abstract state-space with recursive structure, implying the space is infinite.

Focusing on recursive structure as the source of the problem, our course of action is to add a level of indirection, forcing recursive structure to pass through explicitly allocated addresses. Doing so unhinges the recursion in the machine's data structures, enabling structural abstraction via a single point of approximation: the store.

The next section covers the first of the two steps for refactoring the CEK machine into its computable approximation: a store component is introduced to machine states and variable bindings and continuations are redirected through the store. This step introduces no approximation and the constructed machine operates in lock-step with the CEK machine. However, the machine is amenable to a direct structural abstraction.

2.3 CESK* machine

The states of the CESK* machine extend those of the CEK machine to include a *store*, which provides a level of indirection for variable bindings and continuations to pass through. The store is a finite map from *addresses* to *storable values*, which includes closures and continuations, and environments are changed to map variables to addresses. When a variable's value is looked-up by the machine, it is now accomplished by using the environment to look up the variable's address, which is then used to look up the value. To bind a variable to a value, a fresh location in the store is allocated and mapped to the value; the environment is extended to map the variable to that address.

To untie the recursive structure associated with continuations, we likewise add a level of indirection through the store and replace the continuation component of the machine with a *pointer* to a continuation in the store. We term the resulting machine the CESK* (control, environment, store, continuation pointer) machine.

$$\begin{aligned} \varsigma \in \Sigma &= \mathit{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Addr} \\ s \in \mathit{Storable} &= \mathit{Val} \times \mathit{Env} + \mathit{Cont} \\ \kappa \in \mathit{Cont} &= \mathbf{mt} \mid \mathbf{ar}(e, \rho, a) \mid \mathbf{fn}(v, \rho, a). \end{aligned}$$

The transition function for the CESK* machine is defined in Figure 2. The initial state for a program is given by the inj_{CESK^*} function, which combines the expression with the empty environment and a store with a single pointer to the empty continuation, whose address serves as the initial continuation pointer:

$$\mathit{inj}_{CESK^*}(e) = \langle e, \emptyset, [a_0 \mapsto \mathbf{mt}], a_0\rangle.$$

An evaluation function based on this machine is defined following the template of the CEK evaluation given in Section 2.2:

$$\begin{aligned} \mathit{eval}_{CESK^*}(e) = \mathit{real}(v, \rho, \sigma), &\text{ where} \\ &\mathit{inj}_{CESK^*}(e) \longmapsto\!\!\!\!\to_{CESK^*} \langle v, \rho, \sigma, a_0\rangle, \end{aligned}$$

where the *real* function is suitably extended to follow the environment's indirection through the store.

$$\varsigma \longmapsto_{CESK^*} \varsigma', \text{ where } \kappa = \sigma(a), b \notin dom(\sigma)$$

$\langle x, \rho, \sigma, a\rangle$	$\langle v, \rho', \sigma, a\rangle$ where $(v, \rho') = \sigma(\rho(x))$
$\langle (e_0 e_1), \rho, \sigma, a\rangle$	$\langle e_0, \rho, \sigma[b \mapsto \mathbf{ar}(e_1, \rho, a)], b\rangle$
$\langle v, \rho, \sigma, a\rangle$	
if $\kappa = \mathbf{ar}(e, \rho', c)$	$\langle e, \rho', \sigma[b \mapsto \mathbf{fn}(v, \rho, c)], b\rangle$
if $\kappa = \mathbf{fn}((\lambda x.e), \rho', c)$	$\langle e, \rho'[x \mapsto b], \sigma[b \mapsto (v, \rho)], c\rangle$

Figure 2: CESK* machine.

We also define the set of reachable machine states:

$$CESK^*(e) = \{\varsigma \mid inj_{CESK^*}(e) \longmapsto\!\!\!\!\rightarrow_{CESK^*} \varsigma\}.$$

Observe that for any program, the CEK and CESK* machines operate in lock-step: each machine transitions, by the corresponding rule, if and only if the other machine transitions.

Lemma 2 $CESK^*(e) \simeq CEK(e)$

The above lemma implies correctness of the machine.

Lemma 3 (CESK* Correctness) $eval_{CESK^*} = eval$.

Addresses, abstraction and allocation: The CESK* machine, as defined in Figure 2, nondeterministically chooses addresses when it allocates a location in the store, but because machines are identified up to consistent renaming of addresses, the transition system remains deterministic.

Looking ahead, an easy way to bound the state-space of this machine is to bound the set of addresses. But once the store is finite, locations may need to be reused and when multiple values are to reside in the same location; the store will have to soundly approximate this by *joining* the values.

In our concrete machine, all that matters about an allocation strategy is that it picks an unused address. In the abstracted machine however, the strategy *will all but certainly have to re-use previously allocated addresses*. The abstract allocation strategy is therefore crucial to the design of the analysis—it indicates when finite resources should be doled out and decides when information should deliberately be lost in the service of computing within bounded resources. In essence, the allocation strategy is the heart of an analysis.

For this reason, concrete allocation deserves a bit more attention in the machine. An old idea in program analysis is that dynamically allocated storage can be represented by the state of the computation at allocation time [10; 13, Section 1.2.2]. That is, allocation strategies can be based on a (representation) of the machine history. Since machine histories are always fresh, we we call them *time-stamps*.

A common choice for a time-stamp, popularized by Shivers [21], is to represent the history of the computation as *contours*, finite strings encoding the calling context. We present a concrete machine that uses a general time-stamp approach and is parameterized by a choice of *tick* and *alloc* functions.

2.4 Time-stamped CESK* machine

The machine states of the time-stamped CESK* machine include a *time* component, which is intentionally left unspecified:

$$\begin{aligned} t, u &\in Time \\ \varsigma &\in \Sigma = Exp \times Env \times Store \times Addr \times Time. \end{aligned}$$

$$\varsigma \longmapsto_{CESK^*_t} \varsigma', \text{ where } \kappa = \sigma(a), b = alloc(\varsigma), u = tick(\varsigma)$$

$\langle x, \rho, \sigma, a, t\rangle$	$\langle v, \rho', \sigma, a, u\rangle$ where $(v, \rho') = \sigma(\rho(x))$
$\langle (e_0 e_1), \rho, \sigma, a, t\rangle$	$\langle e_0, \rho, \sigma[b \mapsto \mathbf{ar}(e_1, \rho, a)], b, u\rangle$
$\langle v, \rho, \sigma, a, t\rangle$	
if $\kappa = \mathbf{ar}(e, \rho, c)$	$\langle e, \rho, \sigma[b \mapsto \mathbf{fn}(v, \rho, c)], b, u\rangle$
if $\kappa = \mathbf{fn}((\lambda x.e), \rho', c)$	$\langle e, \rho'[x \mapsto b], \sigma[b \mapsto (v, \rho)], c, u\rangle$

Figure 3: Time-stamped CESK* machine.

The machine is parameterized by the functions:

$$tick : \Sigma \to Time \qquad alloc : \Sigma \to Addr.$$

The *tick* function returns the next time; the *alloc* function allocates a fresh address for a binding or continuation. We require of *tick* and *alloc* that for all t and ς, $t \sqsubset tick(\varsigma)$ and $alloc(\varsigma) \notin \sigma$ where $\varsigma = \langle \_, \_, \sigma, \_, t\rangle$.

The time-stamped CESK* machine is defined in Figure 3. Note that occurrences of ς on the right-hand side of this definition are implicitly bound to the state occurring on the left-hand side. The evaluation function $eval_{CESK^*_t}$ and reachable states $CESK^*_t$ are defined following the same outline as before and omitted for space. The initial machine state is defined as:

$$inj_{CESK^*_t}(e) = \langle e, \emptyset, [a_0 \mapsto \mathbf{mt}], a_0, t_0\rangle.$$

Satisfying definitions for the parameters are:

$$Time = Addr = \mathbb{Z}$$

$$a_0 = t_0 = 0 \quad tick\langle \_, \_, \_, \_, t\rangle = t + 1 \quad alloc\langle \_, \_, \_, \_, t\rangle = t.$$

Under these definitions, the time-stamped CESK* machine operates in lock-step with the CESK* machine, and therefore with the CEK machine, implying its correctness.

Lemma 4 $CESK^*_t(e) \simeq CESK^*(e)$.

The time-stamped CESK* machine forms the basis of our abstracted machine in the following section.

2.5 Abstract time-stamped CESK* machine

As alluded to earlier, with the time-stamped CESK* machine, we now have a machine ready for direct abstract interpretation via a single point of approximation: the store. Our goal is a machine that resembles the time-stamped CESK* machine, but operates over a finite state-space and it is allowed to be nondeterministic. Once the state-space is finite, the transitive closure of the transition relation becomes computable, and this transitive closure constitutes a static analysis. Buried in a path through the transitive closure is a possibly infinite traversal that corresponds to the concrete execution of the program.

The abstracted variant of the time-stamped CESK* machine comes from bounding the address space of the store and the number of times available. By bounding the address space, the whole state-space becomes finite. (Syntactic sets like *Exp* are infinite, but finite for any given program.) For the purposes of soundness, an entry in the store may be forced to hold several values simultaneously:

$$\hat{\sigma} \in \widehat{Store} = Addr \to_{\text{fin}} \mathcal{P}(Storable).$$

$\hat{\varsigma} \longmapsto_{\widehat{CESK}^*_t} \hat{\varsigma}'$, where $\kappa \in \hat{\sigma}(a), b = \widehat{alloc}(\hat{\varsigma}, \kappa), u = \widehat{tick}(\hat{\varsigma}, \kappa)$

$\langle x, \rho, \hat{\sigma}, a, t \rangle$	$\langle v, \rho', \hat{\sigma}, a, u \rangle$ where $(v, \rho') \in \hat{\sigma}(\rho(x))$
$\langle (e_0 e_1), \rho, \hat{\sigma}, a, t \rangle$	$\langle e_0, \rho, \hat{\sigma} \sqcup [b \mapsto \mathbf{ar}(e_1, \rho, a)], b, u \rangle$
$\langle v, \rho, \hat{\sigma}, a, t \rangle$	
if $\kappa = \mathbf{ar}(e, \rho', c)$	$\langle e, \rho', \hat{\sigma} \sqcup [b \mapsto \mathbf{fn}(v, \rho, c)], b, u \rangle$
if $\kappa = \mathbf{fn}((\lambda x.e), \rho', c)$	$\langle e, \rho'[x \mapsto b], \hat{\sigma} \sqcup [b \mapsto (v, \rho)], c, u \rangle$

Figure 4: Abstract time-stamped CESK* machine.

Hence, stores now map an address to a *set* of storable values rather than a single value. These collections of values model approximation in the analysis. If a location in the store is re-used, the new value is joined with the current set of values. When a location is dereferenced, the analysis must consider any of the values in the set as a result of the dereference.

The abstract time-stamped CESK* machine is defined in Figure 4. The non-deterministic abstract transition relation changes little compared with the concrete machine. We only have to modify it to account for the possibility that multiple storable values, which includes continuations, may reside together in the store. We handle this situation by letting the machine non-deterministically choose a particular value from the set at a given store location.

The analysis is parameterized by abstract variants of the functions that parameterized the concrete version:

$$\widehat{tick} : \hat{\Sigma} \times \mathit{Cont} \to \mathit{Time}, \quad \widehat{alloc} : \hat{\Sigma} \times \mathit{Cont} \to \mathit{Addr}.$$

In the concrete, these parameters determine allocation and stack behavior. In the abstract, they are the arbiters of precision: they determine when an address gets re-allocated, how many addresses get allocated, and which values have to share addresses.

Recall that in the concrete semantics, these functions consume states—not states and continuations as they do here. This is because in the concrete, a state alone suffices since the state determines the continuation. But in the abstract, a continuation pointer within a state may denote a multitude of continuations; however the transition relation is defined with respect to the choice of a particular one. We thus pair states with continuations to encode the choice.

The *abstract* semantics is given by the reachable states:

$$\widehat{CESK}^*_t(e) = \{\hat{\varsigma} \mid \alpha(inj_{CESK^*_t}(e)) \longmapsto\!\!\!\!\rightarrow_{\widehat{CESK}^*_t} \hat{\varsigma}\}.$$

Soundness and decidability: We have endeavored to evolve the abstract machine gradually so that its fidelity in soundly simulating the original CEK machine is both intuitive and obvious. To formally establish soundness of the abstract time-stamped CESK* machine, we use an abstraction function, defined in Figure 5, from the state-space of the concrete time-stamped machine into the abstracted state-space.

The abstraction map over times and addresses is defined so that the parameters $\widehat{alloc}$ and $\widehat{tick}$ are sound simulations of the parameters *alloc* and *tick*, respectively. We also define the partial order ($\sqsubseteq$) on the abstract state-space as the natural point-wise, element-wise, component-wise and member-wise lifting, wherein the partial orders on the sets *Exp* and *Addr* are flat. Then, we can prove that abstract machine's transition relation simulates the concrete machine's transition relation.

$$
\begin{aligned}
\alpha(e, \rho, \sigma, a, t) &= (e, \alpha(\rho), \alpha(\sigma), \alpha(a), \alpha(t)) && \text{[states]}\\
\alpha(\rho) &= \lambda x.\alpha(\rho(x)) && \text{[environments]}\\
\alpha(\sigma) &= \lambda \hat{a}. \bigsqcup_{\alpha(a)=\hat{a}} \{\alpha(\sigma(a))\} && \text{[stores]}\\
\alpha((\lambda x.e), \rho) &= ((\lambda x.e), \alpha(\rho)) && \text{[closures]}\\
\alpha(\mathbf{mt}) &= \mathbf{mt} && \text{[continuations]}\\
\alpha(\mathbf{ar}(e, \rho, a)) &= \mathbf{ar}(e, \alpha(\rho), \alpha(a))\\
\alpha(\mathbf{fn}(v, \rho, a)) &= \mathbf{fn}(v, \alpha(\rho), \alpha(a))
\end{aligned}
$$

Figure 5: Abstraction map, $\alpha : \Sigma_{CESK^*_t} \to \hat{\Sigma}_{\widehat{CESK}^*_t}$.

Theorem 1 (Soundness)
If $\varsigma \longmapsto_{CEK} \varsigma'$ and $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$, then there exists an abstract state $\hat{\varsigma}'$, such that $\hat{\varsigma} \longmapsto_{\widehat{CESK}^*_t} \hat{\varsigma}'$ and $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$.

Proof. By Lemmas 3 and 4, it suffices to prove soundness with respect to $\longmapsto_{CESK^*_t}$. Assume $\varsigma \longmapsto_{CESK^*_t} \varsigma'$ and $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$. Because ς transitioned, exactly one of the rules from the definition of $(\longmapsto_{CESK^*_t})$ applies. We split by cases on these rules. The rule for the second case is deterministic and follows by calculation. For the remaining (nondeterministic) cases, we must show an abstract state exists such that the simulation is preserved. By examining the rules for these cases, we see that all three hinge on the abstract store in $\hat{\varsigma}$ soundly approximating the concrete store in ς, which follows from the assumption that $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$. □

Theorem 2 (Decidability)
Membership of $\hat{\varsigma}$ in $\widehat{CESK}^*_t(e)$ is decidable.

Proof. The state-space of the machine is non-recursive with finite sets at the leaves on the assumption that addresses are finite. Hence reachability is decidable since the abstract state-space is finite. □

3. ABSTRACT STATE AND CONTROL

We have shown that store-allocated continuations make abstract interpretation of the CESK* machine straightforward. In this section, we want to show that the tight correspondence between concrete and abstract persists after the addition of language features such as conditionals, side effects, and first-class continuations. We tackle each feature, and present the additional machinery required to handle each one. In most cases, the path from a canonical concrete machine to pointer-refined abstraction of the machine is so simple we only show the abstracted system. In doing so, we are arguing that this abstract machine-oriented approach to abstract interpretation represents a flexible and viable framework for building program analyses.

To handle conditionals, we extend the language with a new syntactic form, `(if` e e e`)`, and introduce a base value `#f`, representing false. Conditional expressions induce a new continuation form: $\mathbf{if}(e_0', e_1', \rho, a)$, which represents the evaluation context E[`(if` [ ] e_0 e_1`)`] where ρ closes e_0' to represent e_0, ρ closes e_1' to represent e_1, and a is the address of the representation of E.

$\hat{\varsigma} \longmapsto_{\widehat{CESK}^*_t} \hat{\varsigma}'$, where $\kappa \in \hat{\sigma}(a), b = \widehat{alloc}(\hat{\varsigma}, \kappa), u = \widehat{tick}(\hat{\varsigma}, \kappa)$	
$\langle$`(if` $e_0\ e_1\ e_2$`)`$, \rho, \hat{\sigma}, a, t\rangle$	$\langle e_0, \rho, \hat{\sigma} \sqcup [b \mapsto \mathbf{if}(e_1, e_2, \rho, a)], b, u\rangle$
$\langle$`#f`$, \rho, \hat{\sigma}, a, t\rangle$ if $\kappa = \mathbf{if}(e_0, e_1, \rho', c)$	$\langle e_1, \rho', \hat{\sigma}, c, u\rangle$
$\langle v, \rho, \hat{\sigma}, a, t\rangle$ if $\kappa = \mathbf{if}(e_0, e_1, \rho', c)$, and $v \neq$ `#f`	$\langle e_0, \rho', \hat{\sigma}, c, u\rangle$
$\langle$`(set!` $x\ e$`)`$, \rho, \hat{\sigma}, a, t\rangle$	$\langle e, \rho, \hat{\sigma} \sqcup [b \mapsto \mathbf{set}(\rho(x), a)], b, u\rangle$
$\langle v, \rho, \hat{\sigma}, a, t\rangle$ if $\kappa = \mathbf{set}(a', c)$	$\langle v', \rho, \hat{\sigma} \sqcup [a' \mapsto v], c, u\rangle$ where $v' \in \hat{\sigma}(a')$
$\langle$`(`$\lambda x.e$`)`$, \rho, \hat{\sigma}, a, t\rangle$ if $\kappa = \mathbf{fn}($`callcc`$, \rho', c)$	$\langle e, \rho[x \mapsto b], \hat{\sigma} \sqcup [b \mapsto c], c, u\rangle$ where $c = \widehat{alloc}(\hat{\varsigma}, \kappa)$
$\langle c, \rho, \hat{\sigma}, a, t\rangle$ if $\kappa = \mathbf{fn}($`callcc`$, \rho', a')$	$\langle a, \rho, \hat{\sigma}, c, u\rangle$
$\langle v, \rho, \hat{\sigma}, a, t\rangle$ if $\kappa = \mathbf{fn}(c, \rho', a')$	$\langle v, \rho, \hat{\sigma}, c, u\rangle$

Figure 6: Abstract extended CESK* machine.

Side effects are fully amenable to our approach; we introduce Scheme's `set!` for mutating variables using the `(set!` x e`)` syntax. The `set!` form evaluates its subexpression e and assigns the value to the variable x. Although `set!` expressions are evaluated for effect, we follow Felleisen *et al.* and specify `set!` expressions evaluate to the value of x before it was mutated [7, page 166]. The evaluation context $E[$`(set!` x `[ ])`$]$ is represented by $\mathbf{set}(a_0, a_1)$, where a_0 is the address of x's value and a_1 is the address of the representation of E.

First-class control is introduced by adding a new base value `callcc` which reifies the continuation as a new kind of applicable value. Denoted values are extended to include representations of continuations. Since continuations are store-allocated, we choose to represent them by address. When an address is applied, it represents the application of a continuation (reified via `callcc`) to a value. The continuation at that point is discarded and the applied address is installed as the continuation.

The resulting grammar is:

$$\begin{aligned} e \in \mathit{Exp} &= \ldots \mid \texttt{(if } e\ e\ e\texttt{)} \mid \texttt{(set! } x\ e\texttt{)} \\ \kappa \in \mathit{Cont} &= \ldots \mid \mathbf{if}(e, e, \rho, a) \mid \mathbf{set}(a, a) \\ v \in \mathit{Val} &= \ldots \mid \texttt{\#f} \mid \texttt{callcc} \mid a. \end{aligned}$$

We show only the abstract transitions, which result from store-allocating continuations, time-stamping, and abstracting the concrete transitions for conditionals, mutation, and control. The first three machine transitions deal with conditionals; here we follow the Scheme tradition of considering all non-false values as true. The fourth and fifth transitions deal with mutation.

The remaining three transitions deal with first-class control. In the first of these, `callcc` is being applied to a closure value v. The value v is then "called with the current continuation", *i.e.*, v is applied to a value that represents the continuation at this point. In the second, `callcc` is being applied to a continuation (address). When this value is applied to the reified continuation, it aborts the current computation, installs itself as the current continuation, and puts the reified continuation "in the hole". Finally, in the third, a continuation is being applied; c gets thrown away, and v gets plugged into the continuation b. In all cases, these transitions result from pointer-refinement, time-stamping, and abstraction of the usual machine transitions.

4. ABSTRACT GARBAGE COLLECTION

Garbage collection determines when a store location has become unreachable and can be re-allocated. This is significant in the abstract semantics because an address may be allocated to multiple values due to finiteness of the address space. Without garbage collection, the values allocated to this common address must be joined, introducing imprecision in the analysis (and inducing further, perhaps spurious, computation). By incorporating garbage collection in the abstract semantics, the location may be proved to be unreachable and safely *overwritten* rather than joined, in which case no imprecision is introduced.

Like the rest of the features addressed in this paper, we can incorporate abstract garbage collection into our static analyzers by a straightforward pointer-refinement of textbook accounts of concrete garbage collection, followed by a finite store abstraction.

Concrete garbage collection is defined in terms of a GC machine that computes the reachable addresses in a store [7, page 172]:

$$\langle \mathcal{G}, \mathcal{B}, \sigma\rangle \longmapsto_{GC} \langle (\mathcal{G} \cup LL_\sigma(\sigma(a)) \setminus (\mathcal{B} \cup \{a\})), \mathcal{B} \cup \{a\}, \sigma\rangle$$
$$\text{if } a \in \mathcal{G}.$$

This machine iterates over a set of reachable but unvisited "grey" locations $\mathcal{G}$. On each iteration, an element is removed and added to the set of reachable and visited "black" locations $\mathcal{B}$. Any newly reachable and unvisited locations, as determined by the "live locations" function LL_σ, are added to the grey set. When there are no grey locations, the black set contains all reachable locations. Everything else is garbage.

The live locations function computes a set of locations which may be used in the store. Its definition varies based on the machine being garbage collected, but the definition appropriate for the CESK* machine of Section 2.3 is:

$$\begin{aligned} LL_\sigma(e) &= \emptyset \\ LL_\sigma(e, \rho) &= LL_\sigma(\rho|\mathbf{fv}(e)) \\ LL_\sigma(\rho) &= \mathit{rng}(\rho) \\ LL_\sigma(\mathbf{mt}) &= \emptyset \\ LL_\sigma(\mathbf{fn}(v, \rho, a)) &= \{a\} \cup LL_\sigma(v, \rho) \cup LL_\sigma(\sigma(a)) \\ LL_\sigma(\mathbf{ar}(e, \rho, a)) &= \{a\} \cup LL_\sigma(e, \rho) \cup LL_\sigma(\sigma(a)). \end{aligned}$$

We write $\rho|\mathbf{fv}(e)$ to mean ρ restricted to the domain of free variables in e. We assume the least-fixed-point solution in the calculation of the function LL in cases where it recurs on itself.

The pointer-refinement requires parameterizing the LL function with a store used to resolve pointers to continuations. A nice consequence of this parameterization is that we can re-use LL for *abstract garbage collection* by supplying it an abstract store for the parameter. Doing so only necessitates extending LL to the case of sets of storable values:

$$LL_\sigma(S) = \bigcup_{s \in S} LL_\sigma(s)$$

$$\begin{array}{l|r}
\multicolumn{2}{c}{\varsigma \longmapsto_{CESK^*} \varsigma'} \\
\hline
\langle e, \rho, \sigma, a \rangle & \langle e, \rho, \{\langle b, \sigma(b)\rangle \mid b \in \mathcal{L}\}, a \rangle \\
\multicolumn{2}{l}{\text{if } \langle LL_\sigma(e, \rho) \cup LL_\sigma(\sigma(a)), \{a\}, \sigma \rangle \longmapsto\!\!\!\!\rightarrow_{GC} \langle \emptyset, \mathcal{L}, \sigma \rangle}
\end{array}$$

Figure 7: GC transition for the CESK* machine.

The CESK* machine incorporates garbage collection by a transition rule that invokes the GC machine as a subroutine to remove garbage from the store (Figure 7). The garbage collection transition introduces non-determinism to the CESK* machine because it applies to any machine state and thus overlaps with the existing transition rules. The non-determinism is interpreted as leaving the choice of *when* to collect garbage up to the machine.

The abstract CESK* incorporates garbage collection by the *concrete garbage collection transition*, *i.e.*, we re-use the definition in Figure 7 with an abstract store, $\hat{\sigma}$, in place of the concrete one. Consequently, it is easy to verify abstract garbage collection approximates its concrete counterpart.

The CESK* machine may collect garbage at any point in the computation, thus an abstract interpretation must soundly approximate *all possible choices* of when to trigger a collection, which the abstract CESK* machine does correctly. This may be a useful analysis *of* garbage collection, however it fails to be a useful analysis *with* garbage collection: for soundness, the abstracted machine must consider the case in which garbage is never collected, implying no storage is reclaimed to improve precision.

However, we can leverage abstract garbage collection to reduce the state-space explored during analysis and to improve precision and analysis time. This is achieved (again) by considering properties of the *concrete* machine, which abstract directly; in this case, we want the concrete machine to deterministically collect garbage. Determinism of the CESK* machine is restored by defining the transition relation as a non-GC transition (Figure 2) followed by the GC transition (Figure 7). This state-space of this concrete machine is "garbage free" and consequently the state-space of the abstracted machine is "abstract garbage free."

In the concrete semantics, a nice consequence of this property is that although continuations are allocated in the store, they are deallocated as soon as they become unreachable, which corresponds to when they would be popped from the stack in a non-pointer-refined machine. Thus the concrete machine really manages continuations like a stack.

Similarly, in the abstract semantics, continuations are deallocated as soon as they become unreachable, which often corresponds to when they would be popped. We say often, because due to the finiteness of the store, this correspondence cannot always hold. However, this approach gives a good finite approximation to infinitary stack analyses that can always match calls and returns.

5. RELATED WORK

The study of abstract machines for the λ-calculus began with Landin's SECD machine [11], the systematic construction of machines from semantics with Reynolds's definitional interpreters [20], the theory of abstract interpretation with the seminal work of Cousot and Cousot [4], and static analysis of the λ-calculus with Jones's coupling of abstract machines and abstract interpretation [9]. All have been active areas of research since their inception, but only recently have well known abstract machines been connected with abstract interpretation by Midtgaard and Jensen [14, 15]. We strengthen the connection by demonstrating a general technique for abstracting abstract machines.

The approximation of abstract machine states for the analysis of higher-order languages goes back to Jones [9], who argued abstractions of regular tree automata could solve the problem of recursive structure in environments. We re-invoked that wisdom to eliminate the recursive structure of continuations by allocating them in the store.

Midtgaard and Jensen present a 0CFA for a CPS language [14]. The approach is based on Cousot-style calculational abstract interpretation [3], applied to a functional language. Like the present work, Midtgaard and Jensen start with a known abstract machine for the concrete semantics, the CE machine of Flanagan, *et al.* [8], and employ a reachable-states model. They then compose well-known Galois connections to reveal a 0CFA with reachability in the style of Ayers [1]. The CE machine is not sufficient to interpret direct-style programs, so the analysis is specialized to programs in continuation-passing style.

Although our approach is not calculational like Midtgaard and Jensen's, it continues in their vein by applying abstract interpretation to well known machines, extending the application to direct-style machines to obtain a parameterized family of analyses that accounts for polyvariance.

Static analyzers typically hemorrhage precision in the presence of exceptions and first-class continuations: they jump to the top of the lattice of approximation when these features are encountered. Conversion to continuation- and exception-passing style can handle these features without forcing a dramatic ascent of the lattice of approximation [21]. The cost of this conversion, however, is lost knowledge—both approaches obscure static knowledge of stack structure, by desugaring it into syntax.

Might and Shivers introduced the idea of using abstract garbage collection to improve precision and efficiency in flow analysis [16]. They develop a garbage collecting abstract machine for a CPS language and prove it correct. We extend abstract garbage collection to direct-style languages interpreted on the CESK machine.

6. CONCLUSIONS AND PERSPECTIVE

We have demonstrated a derivational approach to program analysis that yields novel abstract interpretations of languages with higher-order functions, control, state, and garbage collection. These abstract interpreters are obtained by a straightforward pointer refinement and structural abstraction that bounds the address space, making the abstract semantics safe and computable. The technique allows concrete implementation technology, such as garbage collection, to be imported straightforwardly into that of static analysis, bearing immediate benefits. More generally, an abstract machine based approach to analysis shifts the focus of engineering efforts from the design of complex analytic models such as involved constraint languages back to the design of programming languages and machines, from which analysis can be *derived.* Finally, our approach uniformly scales up to richer language features such as laziness, stack-inspection, exceptions, and object-orientation. We speculate that store-allocating bindings and continuations is sufficient for a straightforward abstraction of most existing machines.

Looking forward, a semantics-based approach opens new possibilities for design. Context-sensitive analysis can have daunting complexity [24], which we have made efforts to tame [17], but modular program analysis is crucial to overcome the significant cost of precise abstract interpretation. Modularity can be achieved without needing to design clever approximations, but rather by designing *modular semantics* from which modular analyses follow systematically [23]. Likewise, *push-down analyses* offer infinite state-space abstractions with perfect call-return matching while retaining decidability. Our approach expresses this form of abstraction naturally: the store remains bounded, but continuations stay on the stack.

7. ACKNOWLEDGMENTS

We thank Matthias Felleisen, Jan Midtgaard, Sam Tobin-Hochstadt, and Mitchell Wand for discussions, and the anonymous reviewers of *ICFP'10* for their close reading and helpful critiques; their comments have improved this paper.

8. REFERENCES

[1] A. E. Ayers. *Abstract analysis and optimization of Scheme*. PhD thesis, Massachusetts Institute of Technology, 1993.

[2] M. Biernacka and O. Danvy. A concrete framework for environment machines. *ACM Trans. Comput. Logic*, 9(1):1–30, 2007.

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

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[5] O. Danvy. *An Analytical Approach to Program as Data Objects*. DSc thesis, Department of Computer Science, Aarhus University, Oct. 2006.

[6] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, Department of Computer Science, Aarhus University, Nov. 2004.

[7] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, Aug. 2009.

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

[9] N. D. Jones. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 114–128. Springer-Verlag, 1981.

[10] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 66–74. ACM, 1982.

[11] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[12] P. Meunier, R. B. Findler, and M. Felleisen. Modular set-based analysis from contracts. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 218–231. ACM, Jan. 2006.

[13] J. Midtgaard. Control-flow analysis of functional programs. *ACM Computing Surveys*, 2010.

[14] J. Midtgaard and T. Jensen. A calculational approach to Control-Flow analysis by abstract interpretation. In M. Alpuente and G. Vidal, editors, *SAS*, volume 5079 of *LNCS*, pages 347–362. Springer, 2008.

[15] J. Midtgaard and T. P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 287–298. ACM, 2009.

[16] M. Might and O. 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, Sept. 2006.

[17] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 305–315. ACM, 2010.

[18] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[19] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, Dec. 1975.

[20] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM Annual Conference*, pages 717–740. ACM, 1972.

[21] O. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

[22] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews. *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010.

[23] S. Tobin-Hochstadt and D. V. Horn. Modular analysis via specifications as values. *CoRR*, abs/1103.1362, 2011.

[24] D. Van Horn and H. G. Mairson. Deciding kCFA is complete for EXPTIME. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 275–282. ACM, 2008.

[25] A. K. Wright and S. Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Trans. Program. Lang. Syst.*, 20(1):166–207, 1998.