Systematic Abstraction of Abstract Machines

David Van Horn
Northeastern University

Matthew Might
University of Utah

Abstract

We describe a derivational approach to abstract interpretation that yields novel and transparently sound static analyses when applied to well-established abstract machines for higher-order and imperative programming languages. To demonstrate the technique and support our claim, we transform the CEK machine of Felleisen and Friedman, a lazy variant of Krivine's machine, and the stack-inspecting CM machine of Clements and Felleisen into abstract interpretations of themselves. The resulting analyses bound temporal ordering of program events; predict return-flow and stack-inspection behavior; and approximate the flow and evaluation of by-need parameters. For all of these machines, we find that a series of well-known concrete machine refactorings, plus a technique of store-allocated continuations, leads to machines that abstract into static analyses simply by bounding their stores. We demonstrate that the technique scales up uniformly to allow static analysis of realistic language features, including tail calls, conditionals, side effects, exceptions, first-class continuations, and even garbage collection. In order to close the gap between formalism and implementation, we provide translations of the mathematics as running Haskell code for the initial development of our method.

Introduction

Program analysis aims to soundly predict properties of programs before being run. For over thirty years, the research community has expended significant effort designing effective analyses for higher-order programs (Midtgaard 2011). 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 (Wright and Jagannathan 1998; Nielson et al. 1999; Meunier et al. 2006). 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 runtime, 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 (Landin 1964), and the paths from semantics to machines (Reynolds 1972; Danvy 2006; Felleisen et al. 2009) have a long history in the research on programming languages. From canonical abstract machines such as the CEK machine or Krivine's machine, which represent the idealized core of realistic run-time systems, we perform a series of basic machine refactorings to obtain a *nondeterministic* state-transition system with a *finite* state space. The refactorings are simple: variable bindings and continuations are redirected through the machine's store, and 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 nondeterministic 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.

We demonstrate that the technique allows a direct structural abstraction by bounding the machine's store. (A structural abstraction distributes component-, point-, and member-wise.) 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.

To demonstrate the applicability of the approach, we derive abstract interpreters of:

- a call-by-value λ-calculus with state and control based on the CESK machine of Felleisen and Friedman (Felleisen and Friedman 1987),
- a call-by-need λ-calculus based on a tail-recursive, lazy variant of Krivine's machine derived by Ager, Danvy and Midtgaard (Ager et al. 2004), and
- a call-by-value λ-calculus with stack inspection based on the CM machine of Clements and Felleisen (Clements and Felleisen 2004);

and use abstract garbage collection to improve precision (Might and Shivers 2006).

Finally, we also show that by forgoing stack-allocated continuations, we obtain *push-down* abstract interpretations of programs that form nondeterministic state-transition systems with potentially *infinite* state-spaces. Such abstractions, which constitute recent research breakthroughs (Vardoulakis and Shivers 2011; Earl et al. 2010), precisely match calls to returns and enjoy a natural formulation in our approach.

Overview

In Section 1, we begin with the CEK machine and attempt a structural abstract interpretation, but find ourselves blocked by two recursive structures in the machine: environments and continuations. We make three refactorings to:

1. store-allocate bindings,
2. store-allocate continuations, and
3. time-stamp machine states;

resulting in the CESK, CESK*, and time-stamped CESK* machines, respectively. The time-stamps encode the history (context) of the machine's execution and facilitate context-sensitive abstractions. We then demonstrate that the time-stamped machine abstracts directly into a parameterized, sound and computable static analysis.

In section 2, we instantiate the analysis to obtain a k-CFA-like abstraction and show how to perform store-widening to obtain a polynomial-time 0-CFA abstraction. In Section 3, we replay the abstraction process (slightly abbreviated) with a lazy variant of Krivine's machine to arrive at a static analysis of by-need programs. In Section 4, we incorporate conditionals, side effects, exceptions, and first-class continuations. In Section 5, we show how run-time garbage collection naturally induces a notion of abstract garbage collection, which can improve analysis precision and performance. In Section 6, we abstract the CM (continuation-marks) machine to produce an abstract interpretation of stack inspection.

Background and notation

We assume a basic familiarity with 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* (Felleisen et al. 2009).

1 From CEK to the abstract CESK*

In this section, we start with a traditional machine for a programming language based on the call-by-value λ-calculus, and gradually derive an abstract interpretation of this machine. The outline followed in this section covers the basic steps for systematically deriving abstract interpreters that we follow throughout the rest of the paper.

To begin, consider the following language of expressions:

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

Or, when encoded as an abstract syntax tree in Haskell:

```
type Var    = String
data Lambda = Var :=> Exp
data Exp    = Ref Var
            | Lam Lambda
            | Exp :@ Exp
```

A standard machine for evaluating this language is the CEK machine of Felleisen and Friedman (1986a), and it is from this machine we derive the abstract semantics—a computable approximation of the machine's behavior. Most of the steps in this derivation correspond to well-known machine transformations and real-world implementation techniques—and most of these steps are concerned only with the *concrete machine*; a very simple abstraction is employed only at the very end.

The remainder of this section is outlined as follows: we present the CEK machine, to which we add a store, and use it to allocate variable bindings. This machine is just the CESK machine of Felleisen and Friedman (1987). From here, we further exploit the store to allocate continuations, which corresponds to a well-known implementation technique used in functional language compilers (Shao and Appel 1994). We then abstract *only the store* to obtain a framework for the sound, computable analysis of programs.

1.1 The CEK machine

A standard approach to evaluating programs is to rely on a Curry-Feys-style Standardization Theorem, which says roughly: if an expression e reduces to e' in, *e.g.*, the call-by-value λ-calculus, then e reduces to e' in a canonical manner. This canonical manner thus determines a state machine for evaluating programs: a standard reduction machine.

To define such a machine for our language, we define a grammar of evaluation contexts and notions of reduction (*e.g.*, β_v). An evaluation context is an expression with a "hole" in it. For left-to-right evaluation order, we define evaluation contexts $\mathscr{E}$ as:

$$\mathscr{E} = [\,] \mid (\mathscr{E}\, e) \mid (v\, \mathscr{E}).$$

An expression is either a value or uniquely decomposes into an evaluation context and redex. The standard reduction machine is:

$$\mathscr{E}[e] \longmapsto_{\beta_v} \mathscr{E}[e'], \text{ if } e\ \beta_v\ e'.$$

However, this machine does not shed much light on a realistic implementation. (Accordingly, we will omit a Haskell implementation.) At each step, the machine traverses the entire source of the program looking for a redex. When found, the redex is reduced and the contractum is plugged back in the hole, then the process is repeated.

Abstract machines such as the CEK machine (Felleisen and Friedman 1986b), which are derivable from standard reduction machines, offer an extensionally equivalent but more realistic model of evaluation that is amenable to efficient implementation. The CEK is environment-based; it uses environments and closures to model substitution. It represents evaluation contexts as *continuations*, an inductive data structure that models contexts in an inside-out manner. The key idea of machines such as the CEK is that the whole program need not be traversed to find the next redex, consequently the machine integrates the process of plugging a contractum into a context and finding the next redex.

States of the CEK machine consist of a control string (an expression), an environment that closes the control string, and a continuation:

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

States are identified up to consistent renaming of bound variables.

Environments are finite maps from variables to closures. Environment extension is written $\rho[x \mapsto (v,\rho')]$.

The definition of the state-space in Haskell is similar:

```
type Σ    = (Exp,Env,Cont)
data D    = Clo(Lambda, Env)
type Env  = Var :-> D
data Cont = Mt | Ar(Exp,Env,Cont) | Fn(Lambda,Env,Cont)
```

A notable difference is the need to thread values through a datatype in order to break the unbounded recursion in the type of environments. In this case, datatype `D` contains *d*enotable values. Type operator `:->` is a synonym for the finite map `Data.Map.Map`:

```
type k :-> v = Data.Map.Map k v
```

A little syntactic sugar makes functional extension in Haskell look more like its corresponding formal notation:

```
(==>) :: a -> b -> (a,b)
(==>) x y = (x,y)

(//) :: Ord a => (a :-> b) -> [(a,b)] -> (a :-> b)
(//) f [(x,y)] = Data.Map.insert x y f
```

so that ρ `// [v ==> d]` yields a map identical to ρ except at `v`.

Evaluation contexts $\mathscr{E}$ are represented (inside-out) by continuations as follows: $[\,]$ is represented by **mt**; $\mathscr{E}[(\,]e)]$ is represented by $\mathbf{ar}(e',\rho,\kappa)$ where ρ closes e' to represent e and κ represents $\mathscr{E}$; $\mathscr{E}[(v[\,])]$ is represented by $\mathbf{fn}(v',\rho,\kappa)$ where ρ closes v' to represent v and κ represents $\mathscr{E}$.

The transition function for the CEK machine is defined as follows (we follow the textbook treatment of the CEK machine (Felleisen et al. 2009, page 102)):

$$\begin{array}{ll} \langle x,\rho,\kappa\rangle & \longmapsto_{CEK} \langle v,\rho',\kappa\rangle \text{ where } \rho(x) = (v,\rho') \\ \langle (e_0 e_1),\rho,\kappa\rangle & \longmapsto_{CEK} \langle e_0,\rho,\mathbf{ar}(e_1,\rho,\kappa)\rangle \\ \langle v,\rho,\mathbf{ar}(e,\rho',\kappa)\rangle & \longmapsto_{CEK} \langle e,\rho',\mathbf{fn}(v,\rho,\kappa)\rangle \\ \langle v,\rho,\mathbf{fn}((\lambda x.e),\rho',\kappa)\rangle & \longmapsto_{CEK} \langle e,\rho'[x \mapsto (v,\rho)],\kappa\rangle \end{array}$$

Now, we have to render the transition relation $\longmapsto \subseteq \Sigma \times \Sigma$ as code. There are many ways to render a relation $R \subseteq A \times B$ in code. For finite relations, we could construct R as a set of pairs. For infinite relations, we could render R as a predicate:

$$R \cong A \times B \to \mathit{Boolean},$$

or as a function:

$$R \cong A \to \mathscr{P}(B).$$

In the Haskell implementation, we render the transition relation as a function, `step`:

```
step :: Σ -> Σ
step (Ref x, ρ, κ) = (Lam lam,ρ',κ) where Clo(lam, ρ') = ρ!x
step (f :@ e, ρ, κ) = (f, ρ,  Ar(e, ρ, κ))
step (Lam lam, ρ, Ar(e, ρ', κ)) = (e, ρ', Fn(lam, ρ, κ))
step (Lam lam, ρ, Fn(x :=> e, ρ', κ)) = (e, ρ' // [x ==> Clo(lam, ρ)], κ)
```

Since the transition relation is deterministic, we do not expand the range of this function to a set. The initial machine state for a closed expression e is given by the *inj* function:

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

In Haskell, the injection function is almost identical:

```
inject :: Exp -> Σ
inject (e) = (e, Data.Map.empty, Mt)
```

Typically, an evaluation function is defined as a partial function from closed expressions to answers:

$$eval'_{CEK}(e) = (v, \rho) \text{ if } inj(e) \longmapsto\!\!\!\!\!\rightarrow_{CEK} \langle v, \rho, \mathbf{mt} \rangle.$$

This gives an extensional view of the machine, which is useful, *e.g.*, to prove correctness with respect to a canonical evaluation function such as one defined by standard reduction or compositional valuation. However for the purposes of program analysis, we are concerned more with the intensional aspects of the machine. As such, we define the meaning of a program as the (possibly infinite) set of reachable machine states:

$$eval_{CEK}(e) = \{\varsigma \mid inj(e) \longmapsto\!\!\!\!\!\rightarrow_{CEK} \varsigma\}.$$

In Haskell, we can use a `collect` auxiliary function:

```
collect :: (a -> a) -> (a -> Bool) -> a -> [a]
collect f isFinal ς0 | isFinal ς0 = [ς0]
                     | otherwise  = ς0:(collect f isFinal (f(ς0)))
```

to define `evaluate`:

```
evaluate :: Exp -> [Σ]
evaluate e = collect step isFinal (inject(e))
```

where the `isFinal` function watches for proper final states:

```
isFinal :: Σ -> Bool
isFinal (Lam _, ρ, Mt) = True
isFinal _              = False
```

An outline for abstract interpretation Deciding membership in the set of reachable machine states is not possible due to the halting problem. The goal of abstract interpretation, then, is to construct a function, $aval_{\widehat{CEK}}$, that is a sound and computable approximation to the $eval_{CEK}$ 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 $(\longmapsto_{\widehat{CEK}}) \subseteq \hat{\Sigma} \times \hat{\Sigma}$, which operates over

abstract states, $\hat{\Sigma}$, which approximate the states of the CEK machine, and an abstraction map $\alpha : \Sigma \to \hat{\Sigma}$ that maps concrete machine states into abstract machine states.

The abstract evaluation function is then defined as:

$$aval_{\widehat{CEK}}(e) = \{\hat{\varsigma} \mid \alpha(inj(e)) \longmapsto\!\!\!\!\!\rightarrow_{\widehat{CEK}} \hat{\varsigma}\}.$$

1. We achieve *decidability* by constructing the approximation in such a way that the state-space of the abstracted machine is finite, which guarantees that for any closed expression *e*, the set $aval(e)$ is finite.
2. We achieve *soundness* by demonstrating the abstracted machine transitions preserve the abstraction map, so that if $\varsigma \longmapsto \varsigma'$ and $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$, then there exists an abstract state $\hat{\varsigma}'$ such that $\hat{\varsigma} \longmapsto \hat{\varsigma}'$ and $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$.

1.2 A first attempt at abstract interpretation

A simple approach to abstracting the machine's state-space is to apply a *structural abstract interpretation*, which lifts abstraction point-wise, element-wise, component-wise and member-wise 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 map α yields objects in an abstract state-space with recursive structure, implying the space is infinite. It is possible to perform abstract interpretation over an infinite state-space, but it requires a widening operator. A widening operator accelerates the ascent up the lattice of approximation and must guarantee convergence. It is difficult to imagine a widening operator, other than the one that jumps immediately to the top of the lattice, for these semantics.

Focusing on recursive structure as the source of the problem, a reasonable course of action is to add a level of indirection to the recursion—to force recursive structure to pass through explicitly allocated addresses. In doing so, we will unhinge recursion in a program's data structures and its control-flow from recursive structure in the state-space.

We turn our attention next to the CESK machine (Felleisen 1987; Felleisen and Friedman 1987), since the CESK machine eliminates recursion from one of the structures in the CEK machine: environments. In the subsequent section (Section 1.5), we will develop a CESK machine with a pointer refinement (CESK*) that eliminates the other source of recursive structure: continuations. At that point, the machine structurally abstracts via a single point of approximation: the store.

1.3 The 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 to pass through. The store is a finite map from *addresses* to *storable values* 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.

The state-space for the CESK machine is defined as follows:

$$\begin{array}{rl} \varsigma \in \Sigma & = Exp \times Env \times Store \times Cont \\ \rho \in Env & = Var \to_{\text{fin}} Addr \\ \sigma \in Store & = Addr \to_{\text{fin}} Storable \\ s \in Storable & = Lam \times Env \\ a,b,c \in Addr & \text{an infinite set.} \end{array}$$

States are identified up to consistent renaming of bound variables and addresses. In Haskell:

```
type Σ = (Exp,Env,Store,Kont)
type Env = Var :-> Addr
data Storable = Clo (Lambda, Env)
type Store = Addr :-> Storable
data Kont = Mt | Ar(Exp,Env,Kont) | Fn(Lambda,Env,Kont)
type Addr = Int
```

The transition function for the CESK machine is defined as follows (we follow the textbook treatment of the CESK machine (Felleisen et al. 2009, page 166)):

$$\begin{array}{ll} \langle x,\rho,\sigma,\kappa\rangle & \longmapsto_{CESK} \langle v,\rho',\sigma,\kappa\rangle \text{ where } \sigma(\rho(x)) = (v,\rho') \\ \langle (e_0 e_1),\rho,\sigma,\kappa\rangle & \longmapsto_{CESK} \langle e_0,\rho,\sigma,\mathbf{ar}(e_1,\rho,\kappa)\rangle \\ \langle v,\rho,\sigma,\mathbf{ar}(e,\rho',\kappa)\rangle & \longmapsto_{CESK} \langle e,\rho',\sigma,\mathbf{fn}(v,\rho,\kappa)\rangle \\ \langle v,\rho,\sigma,\mathbf{fn}((\lambda x.e),\rho',\kappa)\rangle & \longmapsto_{CESK} \langle e,\rho'[x\mapsto a],\sigma[a\mapsto(v,\rho)],\kappa\rangle \text{ where } a \notin dom(\sigma) \end{array}$$

In Haskell, the transition relation is once again a function:

```
step :: Σ -> Σ
step (Ref x, ρ, σ, κ) = (Lam lam, ρ', σ, κ)
  where Clo (lam, ρ') = σ!(ρ!x)
step (f :@ e, ρ, σ, κ) = (f, ρ, σ,  Ar(e, ρ, κ))
step (Lam lam,ρ,σ,Ar(e, ρ', κ)) = (e, ρ', σ, Fn(lam, ρ, κ))
step (Lam lam,ρ,σ,Fn(x :=> e, ρ', κ)) =
  (e, ρ' // [x ==> a'], σ // [a' ==> Clo (lam, ρ)], κ)
    where a' = alloc(σ)
```

A key difference is that instead of choosing any address not currently in the store for binding variables, we require a well-defined process for choosing a free address. For that, we use the `alloc` function:

```
alloc :: Store -> Addr
alloc(σ) = (foldl max 0 $! keys σ) + 1
```

The initial state for a closed expression is given by the *inj* function, which combines the expression with the empty environment, store, and continuation:

$$inj_{CESK}(e) = \langle e,\emptyset,\emptyset,\mathbf{mt}\rangle.$$

In Haskell:

```
inject :: Exp -> Σ
inject (e) = (e, ρ0, σ0, Mt)
 where ρ0 = Data.Map.empty
       σ0 = Data.Map.empty
```

The $eval_{CESK}$ evaluation function is defined following the template of the CEK evaluation given in Section 1.1:

$$eval_{CESK}(e) = \{\varsigma \mid inj(e) \longmapsto\!\!\!\!\!\rightarrow_{CESK} \varsigma\},$$

which is identical in Haskell to the prior version, except for the final-state recognizer:

```
isFinal :: Σ -> Bool
isFinal (Lam _, _, _, Mt) = True
isFinal _                 = False
```

Observe that for any closed expression, 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 1 (*Felleisen 1987*)
$eval_{CESK}(e) \simeq eval_{CEK}(e)$.

1.4 A second attempt at abstract interpretation

With the CESK machine, half the problem with the attempted naïve abstract interpretation is solved: environments and closures are no longer mutually recursive. Unfortunately, continuations still have recursive structure. We could crudely abstract a continuation into a set of frames, losing all sense of order, but this would lead to a static analysis lacking faculties to reason about return-flow: every call would appear to return to every other call. A better solution is to refactor continuations as we did environments, redirecting the recursive structure through the store. In the next section, we explore a CESK machine with a pointer refinement for continuations.

1.5 The CESK* machine

To untie the recursive structure associated with continuations, we shift to store-allocated continuations. The basic idea behind store-allocated continuations is not new. SML/NJ has allocated continuations in the heap for well over a decade (Shao and Appel 1994). At first glance, modeling the program stack in an abstract machine with store-allocated continuations would not seem to provide any real benefit. Indeed, for the purpose of defining the meaning of a program, there is no benefit, because the meaning of the program does not depend on the stack-implementation strategy. Yet, a closer inspection finds that store-allocating continuations eliminate recursion from the definition of the state-space of the machine. With no recursive structure in the state-space, an abstract machine becomes eligible for conversion into an abstract interpreter through a simple structural abstraction.

States of the CESK* machine, like the CESK, consist of an expression, environment, store, and continuation; however, continuations are represented slightly differently. Instead of the inductive definition of continuations as

$$\kappa \in Cont = \mathbf{mt} \mid \mathbf{ar}(e, \rho, \kappa) \mid \mathbf{fn}(v, \rho, \kappa),$$

we insert a level of indirection by replacing the continuation of a frame with *a pointer to a continuation*:

$$\kappa \in Cont = \mathbf{mt} \mid \mathbf{ar}(e, \rho, a) \mid \mathbf{fn}(v, \rho, a).$$

This change requires the store to follow suit by mapping addresses to denotable values or continuations:

$$s \in \mathit{Storable} = \mathit{Val} \times \mathit{Env} + \mathit{Cont}.$$

All together, the new state-space in Haskell becomes:

```
type Σ = (Exp,Env,Store,Kont)
data Kont = Mt | Ar(Exp,Env,Addr) | Fn(Lambda,Env,Addr)
data Storable = Clo(Lambda, Env) | Cont Kont
type Env = Var :-> Addr
type Store = Addr :-> Storable
type Addr = Int
```

The revised machine is defined as

$$\begin{array}{ll}
\langle x,\rho,\sigma,\kappa\rangle & \longmapsto_{CESK^\star} \langle v,\rho',\sigma,\kappa\rangle \text{ where } (v,\rho') = \sigma(\rho(x)) \\
\langle (e_0 e_1),\rho,\sigma,\kappa\rangle & \longmapsto_{CESK^\star} \langle e_0,\rho,\sigma[a \mapsto \kappa],\mathbf{ar}(e_1,\rho,a)\rangle \text{ where } a \notin \mathit{dom}(\sigma) \\
\langle v,\rho,\sigma,\mathbf{ar}(e,\rho',a)\rangle & \longmapsto_{CESK^\star} \langle e,\rho',\sigma,\mathbf{fn}(v,\rho,a)\rangle \\
\langle v,\rho,\sigma,\mathbf{fn}((\lambda x.e),\rho',b)\rangle & \longmapsto_{CESK^\star} \langle e,\rho'[x \mapsto a],\sigma[a \mapsto (v,\rho)],\kappa\rangle \\
 & \quad \text{where } a \notin \mathit{dom}(\sigma) \text{ and } \kappa = \sigma(b)
\end{array}$$

and the initial machine state is defined just as before:

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

In Haskell:

```
step :: Σ -> Σ
step (Ref x, ρ, σ, κ) = (Lam lam, ρ', σ, κ)
 where Clo(lam, ρ') = σ!(ρ!x)
step (f :@ e, ρ, σ, κ) = (f, ρ, σ', κ')
 where a' = alloc(σ)
       σ' = σ // [a' ==> Cont κ]
       κ' = Ar(e, ρ, a')
step (Lam lam, ρ, σ, Ar(e, ρ', a')) = (e, ρ', σ, Fn(lam, ρ, a'))
step (Lam lam, ρ, σ, Fn(x :=> e, ρ', a)) =
     (e, ρ' // [x ==> a'], σ // [a' ==> Clo(lam, ρ)], κ)
 where Cont κ = σ!a
       a' = alloc(σ)
```

The allocation function needs only to return an unused address:

```
alloc :: Store -> Addr
alloc(σ) = (foldl max 0 $! keys σ) + 1
```

The evaluation function (not shown) is defined along the same lines as those for the CEK (Section 1.1) and CESK (Section 1.3) machines. Like the CESK machine, it is easy to relate the CESK* machine to its predecessor; from corresponding initial configurations, these machines operate in lock-step:

Lemma 2
$\mathit{eval}_{CESK^\star}(e) \simeq \mathit{eval}_{CESK}(e)$.

1.6 Addresses, abstraction and allocation

The CESK* machine 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.[1] 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 *may 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. Allocation strategies corresponding to well-known analyses are given in Section 2.

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 (Jones and Muchnick 1982; Midtgaard 2011, Section 1.2.2). That is, allocation strategies can be based on a (representation) of the machine history. These representations are often called *time-stamps*.

A common choice for a time-stamp, popularized by Shivers (1991), is to represent the history of the computation as *contours*, finite strings encoding the calling context. We present a concrete machine that uses general time-stamp approach and is parameterized by a choice of $tick$ and *alloc* functions. We then instantiate $tick$ and *alloc* to obtain an abstract machine for computing a k-CFA-style analysis using the contour approach.

1.7 The time-stamped CESK* machine

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

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

In Haskell, we fix times and addresses as integers for the moment:

```
type Σ = (Exp,Env,Store,Kont,Time)
data Storable = Clo (Lambda, Env) | Cont Kont
type Env = Var :-> Addr
type Store = Addr :-> Storable
data Kont = Mt | Ar (Exp,Env,Addr) | Fn (Lambda,Env,Addr)
type Addr = Int
type Time = Int
```

[1] A finite number of addresses leads to a finite number of environments, which leads to a finite number of closures and continuations, which in turn, leads to a finite number of stores, and finally, a finite number of states.

The machine is parameterized by the functions:

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

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 $\varsigma = \langle \_, \_, \sigma, \_, t \rangle$, $t \sqsubset tick(\varsigma)$ and $alloc(\varsigma) \notin \sigma$. In Haskell, these functions find the next available integer:

```
alloc :: Σ -> Addr
alloc(_,_,σ,_,_) = (foldl max 0 $! keys σ) + 1

tick :: Σ -> Time
tick (_,_,_,_,t) = t + 1
```

The time-stamped CESK* machine transition relation, $\varsigma \longmapsto_{CESK^\star} \varsigma'$, is defined as:

$$\begin{array}{ll}
\langle x, \rho, \sigma, \kappa, t \rangle & \longmapsto_{CESK^\star_t} \langle v, \rho', \sigma, \kappa, u \rangle \text{ where } (v, \rho') = \sigma(\rho(x)) \\
\langle (e_0 e_1), \rho, \sigma, \kappa, t \rangle & \longmapsto_{CESK^\star_t} \langle e_0, \rho, \sigma[a \mapsto \kappa], \mathbf{ar}(e_1, \rho, a), u \rangle \\
\langle v, \rho, \sigma, \mathbf{ar}(e, \rho', c), t \rangle & \longmapsto_{CESK^\star_t} \langle e, \rho', \sigma, \mathbf{fn}(v, \rho, c), u \rangle \\
\langle v, \rho, \sigma, \mathbf{fn}((\lambda x.e), \rho', c), t \rangle & \longmapsto_{CESK^\star_t} \langle e, \rho'[x \mapsto a], \sigma[a \mapsto (v, \rho)], \kappa, u \rangle \text{ where } \kappa = \sigma(c)
\end{array}$$

where $a = alloc(\varsigma)$ and $u = tick(\varsigma)$. Or, in Haskell:

```
step :: Σ -> Σ
step ς@(Ref x, ρ, σ, κ, t) = (Lam lam, ρ', σ, κ, t')
 where Clo(lam, ρ') = σ!(ρ!x)
       t' = tick(ς)

step ς@(f :@ e, ρ, σ, κ, t) = (f, ρ, σ', κ', t')
 where a' = alloc(ς)
       σ' = σ // [a' ==> Cont κ]
       κ' = Ar(e, ρ, a')
       t' = tick(ς)

step ς@(Lam lam, ρ, σ, Ar(e, ρ', a'), t)
     = (e, ρ', σ, Fn(lam, ρ, a'), t')
 where t' = tick(ς)

step ς@(Lam lam, ρ, σ, Fn(x :=> e, ρ', a), t)
     = (e, ρ' // [x ==> a'], σ // [a' ==> Clo(lam, ρ)], κ, t')
 where Cont κ = σ!a
       a' = alloc(ς)
       t' = tick(ς)
```

A program is injected into the initial machine state as:

$$inj_{CESK^\star_t}(e) = \langle e, \emptyset, \emptyset, \mathbf{mt}, t_0 \rangle.$$

Satisfying definitions for the parameters are:

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

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

Under these definitions, the time-stamped CESK* machine operates in lock-step with the CESK* machine, and therefore with the CESK and CEK machines as well.

Lemma 3
$eval_{CESK^\star_t}(e) \simeq eval_{CESK^\star}(e)$.

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

1.8 The 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 these sets, the state-space becomes finite,[2] but for the purposes of soundness, an entry in the store may be forced to hold several values simultaneously:

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

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. In Haskell, the new state-space is nearly the same:

```
type Σ = (Exp,Env,Store,Kont,Time)
data Storable = Clo(Lambda, Env) | Cont Kont
type Env = Var :-> Addr
type Store = Addr :-> ℙ(Storable)
data Kont = Mt | Ar(Exp,Env,Addr) | Fn(Lambda,Env,Addr)
type Time = -- some finite set
type Addr = -- some finite set
```

where $\mathbb{P}$ is a type synonym for `Data.Set.Set`:

```
type ℙ s = Data.Set.Set s
```

[2] Syntactic sets like *Exp* are infinite, but finite for any given program.

The nondeterministic 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, which we handle by letting the machine nondeterministically choose a particular value from the set at a given store location.

The abstract time-stamped CESK* machine is defined as:

$$\begin{array}{ll}
\langle x, \rho, \hat{\sigma}, \kappa, t\rangle & \longmapsto_{\widehat{CESK^\star_t}} \langle v, \rho', \hat{\sigma}, \kappa, u\rangle \text{ where } (v, \rho') \in \hat{\sigma}(\rho(x)) \\
\langle (e_0 e_1), \rho, \hat{\sigma}, \kappa, t\rangle & \longmapsto_{\widehat{CESK^\star_t}} \langle e_0, \rho, \hat{\sigma} \sqcup [a \mapsto \{\kappa\}], \mathbf{ar}(e_1, \rho, a), u\rangle \\
\langle v, \rho, \hat{\sigma}, \mathbf{ar}(e, \rho', c), t\rangle & \longmapsto_{\widehat{CESK^\star_t}} \langle e, \rho', \hat{\sigma}, \mathbf{fn}(v, \rho, c), u\rangle \\
\langle v, \rho, \hat{\sigma}, \mathbf{fn}((\lambda x.e), \rho', c), t\rangle & \longmapsto_{\widehat{CESK^\star_t}} \langle e, \rho'[x \mapsto a], \hat{\sigma} \sqcup [a \mapsto \{(v, \rho)\}], \kappa, u\rangle \text{ where } \kappa \in \hat{\sigma}(c)
\end{array}$$

where $a = \widehat{alloc}(\hat{\varsigma})$ and $u = \widehat{tick}(\hat{\varsigma})$. To make sense of the join operator $\sqcup$, we assume the natural lifting of a partial order over sets and maps.

Haskell requires that we be explicit about the "natural" lifting. Fortunately, we can specify the natural lifting through type classes. First, we define a class for lattices, sets partially ordered by a relation $\sqsubseteq$, and for which any two elements have both a least upper bound ($\sqcup$) and a greatest lower bound ($\sqcap$):

```
class Lattice a where
 bot :: a
 top :: a
 (⊑) :: a -> a -> Bool
 (⊔) :: a -> a -> a
 (⊓) :: a -> a -> a
```

Then, we assert that for a flat set X ordered by equality, the set $\mathscr{P}(X)$ is a lattice ordered by set inclusion:

```
instance (Ord s, Eq s) => Lattice (ℙ s) where
 bot = Data.Set.empty
 top = error "no representation of universal set"
 x ⊔ y = x `Data.Set.union` y
 x ⊓ y = x `Data.Set.intersection` y
 x ⊑ y = x `Data.Set.isSubsetOf` y
```

This allows us to treat sets of *Storable* objects as a lattice. Next, we lift maps into lattices point-wise into lattices:

```
instance (Ord k, Lattice v) => Lattice (k :-> v) where
 bot = Data.Map.empty
 top = error "no representation of top map"
 f ⊑ g = Data.Map.isSubmapOfBy (⊑) f g
 f ⊔ g = Data.Map.unionWith (⊔) f g
 f ⊓ g = Data.Map.intersectionWith (⊓) f g
```

To provide the illusion of infinite maps, we also define a new look-up operator that returns the bottom element of the range by default:

```
(!!) :: (Ord k, Lattice v) => (k :-> v) -> k -> v
f !! k = Data.Map.findWithDefault bot k f
```

At this point, abstract stores are now lattices with a sensibly defined join operation $\sqcup$:

$$\hat{\sigma}_1 \sqcup \hat{\sigma}_2 = \lambda a.\hat{\sigma}_1(a) \sqcup \hat{\sigma}_2(a).$$

To render the transition relation in code requires lifting the range of the `step` function to a sequence, since the abstract relation is truly nondeterministic:

```
step :: Σ -> [Σ]
step ς@(Ref x, ρ, σ, κ, t) = [ (Lam lam, ρ', σ, κ, t')
  | Clo(lam, ρ') <- Data.Set.toList $! σ!!(ρ!x) ]
  where t' = tick(ς)

step ς@(f :@ e, ρ, σ, κ, t) = [ (f, ρ, σ', κ', t') ]
 where a' = alloc(ς)
       σ' = σ ⨆ [a' ==> s(Cont κ)]
       κ' = Ar(e, ρ, a')
       t' = tick(ς)

step ς@(Lam lam, ρ, σ, Ar(e, ρ', a'), t)
     = [ (e, ρ', σ, Fn(lam, ρ, a'), t') ]
 where t' = tick(ς)

step ς@(Lam lam, ρ, σ, Fn(x :=> e, ρ', a), t)
     = [ (e, ρ' // [x ==> a'], σ ⨆ [a' ==> s(Clo(lam, ρ))], κ, t')
       | Cont κ <- Data.Set.toList $! σ!!a ]
 where t' = tick(ς)
       a' = alloc(ς)
```

For convenience, we overrode the big join operator $\bigsqcup$ to serve as a special operator for merging a few entries into a large map lattice:

```
(⨆) :: (Ord k, Lattice v) => (k :-> v) -> [(k,v)] -> (k :-> v)
f ⨆ [(k,v)] = Data.Map.insertWith (⊔) k v f
```

and made `s` a synonym for `singleton`:

```
s x = Data.Set.singleton x
```

A program is injected into the initial abstract machine state just as before:

$$\widehat{inj_{CESK_t^\star}}(e) = inj_{CESK_t^\star}(e) = \langle e, \emptyset, \emptyset, \mathbf{mt}, t_0 \rangle.$$

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

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

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 computes the set of reachable states:

$$aval_{\widehat{CESK^\star_t}}(e) = \{\hat{\varsigma} \mid inj_{\widehat{CESK^\star_t}}(e) \longmapsto\!\!\!\!\!\rightarrow_{\widehat{CESK^\star_t}} \hat{\varsigma}\}.$$

In Haskell, computing the analysis is (naively) just a graph exploration:

```
aval :: Exp -> ℙ(Σ)
aval(e) = explore step (inject(e))

explore :: (Ord a) => (a -> [a]) -> a -> ℙ(a)
explore f ς0 = search f Data.Set.empty [ς0]

(∈) :: Ord a => a -> ℙ(a) -> Bool
(∈) = Data.Set.member

search :: (Ord a) => (a -> [a]) -> ℙ(a) -> [a] -> ℙ(a)
search f seen [] = seen
search f seen (hd:tl)
 | hd ∈ seen = search f seen tl
 | otherwise = search f (Data.Set.insert hd seen) (f(hd) ++ tl)
```

1.9 Soundness and computability

The finiteness of the abstract state-space ensures decidability.

Theorem 1 (*Decidability of the Abstract CESK⋆ Machine*)

$\hat{\varsigma} \in aval_{\widehat{CESK^\star_t}}(e)$ 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. □

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. But to formally establish soundness of the abstract time-stamped CESK⋆ machine, we use an abstraction function, defined below, from the state-space of the concrete time-stamped machine into

the abstracted state-space.

$$
\begin{aligned}
\alpha : \Sigma_{CESK^\star_t} &\to \hat{\Sigma}_{\widehat{CESK^\star_t}} \\
\alpha(e,\rho,\sigma,a,t) &= (e, \alpha(\rho), \alpha(\sigma), \alpha(\kappa), \alpha(t)) && \text{[states]} \\
\alpha(\rho) &= \lambda x.\alpha(\rho(x)) && \text{[environments]} \\
\alpha(\sigma) &= \lambda \hat{a}. \bigsqcup_{\alpha(a)=\hat{a}} \{\alpha(\sigma(a))\} && \text{[stores]} \\
\alpha((\lambda x.e),\rho) &= ((\lambda x.e), \alpha(\rho)) && \text{[closures]} \\
\alpha(\mathbf{mt}) &= \mathbf{mt} && \text{[continuations]} \\
\alpha(\mathbf{ar}(e,\rho,a)) &= \mathbf{ar}(e, \alpha(\rho), \alpha(a)) \\
\alpha(\mathbf{fn}(v,\rho,a)) &= \mathbf{fn}(v, \alpha(\rho), \alpha(a)),
\end{aligned}
$$

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.

Theorem 2 (*Soundness of the Abstract CESK* Machine*)

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

Proof

By Lemmas 1, 2, and 3, it suffices to prove soundness with respect to $\longmapsto_{CESK^\star_t}$. Assume $\varsigma \longmapsto_{CESK^\star_t} \varsigma'$ and $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$. Because ς transitioned, exactly one of the rules from the definition of $(\longmapsto_{CESK^\star_t})$ applies. We split by cases on these rules. The rule for the second case is deterministic and follows by calculation. For the 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 $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$. $\square$

2 An approximation like *k*-CFA

In this section, we instantiate the time-stamped CESK* machine to obtain a contour-based machine; this instantiation forms the basis of a context-sensitive abstract interpreter with polyvariance like that found in *k*-CFA (Shivers 1991). In preparation for abstraction, we first refine the time-stamped machine to link the allocation of times and addresses. Under abstraction, this link defines the relationship between *context-sensitivity* and *polyvariance* in static analysis.

2.1 A machine with time-based allocation

We can take the last concrete machine and refine it so that the allocation of times and addresses are linked. We do so by creating two kinds of addresses: variable binding addresses and continuation addresses:

$$Addr = \overbrace{Var \times Time}^{\text{binding addr.}} + \overbrace{Exp \times Time}^{\text{cont. addr.}}$$

When a variable is stored, the address it receives is a combination of itself and the time of its binding. When a continuation is stored, the address it receives is a combination of the expression forcing the creation of the continuation plus the time of its creation. In both cases, the freshness of the time ensures the freshness of the address.

With all the domains together in Haskell:

```
type Σ = (Exp,Env,Store,Kont,Time)
data Storable = Clo (Lambda, Env) | Cont Kont
type Env = Var :-> Addr
type Store = Addr :-> Storable
data Kont = Mt | Ar (Exp,Env,Addr) | Fn (Lambda,Env,Addr)
type Time = Int
data Addr = KAddr (Exp, Time)
          | BAddr (Var, Time)
```

The formal concrete semantics do not change with this machine. In Haskell, however, it helps to split allocation into two functions—one that allocates addresses for variables, and the other for continuations:

```
allocBind :: (Var,Time) -> Addr
allocBind (v,t) = BAddr (v,t)

allocKont :: (Exp,Time) -> Addr
allocKont (e,t) = KAddr (e,t)
```

so that the `step` function invokes each as appropriate:

```
step :: Σ -> Σ
step ς@(Ref x, ρ, σ, κ, t) = (Lam lam, ρ', σ, κ, t')
 where Clo(lam, ρ') = σ!(ρ!x)
       t' = tick(ς)

step ς@(f :@ e, ρ, σ, κ, t) = (f, ρ, σ', κ', t')
 where a' = allocKont(f :@ e, t')
       σ' = σ // [a' ==> Cont κ]
       κ' = Ar(e, ρ, a')
       t' = tick(ς)

step ς@(Lam lam, ρ, σ, Ar(e, ρ', a'), t)
     = (e, ρ', σ, Fn(lam, ρ, a'), t')
 where t' = tick(ς)

step ς@(Lam lam, ρ, σ, Fn(x :=> e, ρ', a), t)
     = (e, ρ' // [x ==> a'], σ // [a' ==> Clo(lam, ρ)], κ, t')
 where Cont κ = σ!a
       a' = allocBind(x, t')
       t' = tick(ς)
```

2.2 Instantiating time as context

Up to this point, we have left time opaque (or used the integers in Haskell). In this section, we will change the structure of time so as to (1) encode execution context, and (2) make it more easily abstractable.

Call strings have long served as a measure of execution contexts in program analysis (Sharir and Pnueli 1981). To take this approach in the abstract machine framework, we set time to the sequence of expressions seen since the start of execution:

$$Time = Exp^*.$$

Then, we modify the *tick* function to prepend the current expression:

$$tick\langle e, \_, \_, \_, t\rangle = e : t$$

Of course, this definition captures *expression* strings rather than *call* strings. Call strings are recoverable by ignoring the non-application terms in the sequence.

In Haskell, only the definition of the type `Time` and the function `tick` change:

```
type Time = [Exp]

tick :: Σ -> Time
tick (e,_,_,_,t) = e : t
```

2.3 A machine for k-CFA-like approximation

Bounding the length of the time in the previous machine to at most k and then apply the abstraction process yields a k-CFA-like machine.

Formally, the *tick* function restricts itself to the last k call sites:

$$tick\langle e, \_, \_, \_, t\rangle = \lfloor e : t \rfloor_k$$

or, in Haskell:

```
tick :: Σ -> Time
tick (e,_,_,_,t) = take k (e : t)
```

Comparison to k-CFA: We say "k-CFA-like" rather than "k-CFA" because there are distinctions between the machine just described and k-CFA:

1. k-CFA focuses on "what flows where"; the ordering between states in the abstract transition graph produced by our machine produces "what flows where *and when*."
2. Standard presentations of k-CFA implicitly inline a global approximation of the store into the algorithm (Shivers 1991); ours uses one store per state to increase precision at the cost of complexity. We can explicitly inline the store to achieve the same complexity, as shown in Section 2.5.
3. On function call, k-CFA merges argument values together with previous instances of those arguments from the same context; our "minimalist" evolution of the abstract machine takes a higher-precision approach: it forks the machine for each argument value, rather than merging them immediately.
4. k-CFA does not recover explicit information about stack structure; our machine contains an explicit model of the stack for every machine state.

2.4 A machine for 0-CFA-like approximation

Let $k = 0$. Notice that $\widehat{Time}$ collapses to a constant, and $\widehat{Addr}$ collapses to variables and expressions. Since time-stamps have collapsed, they may be eliminated from the machine entirely:

$$\widehat{Addr} = Exp + Var$$

By in-lining the allocation function and observing environments in the in-lined 0-CFA machine are always the identity environment, they can be eliminated, we obtain a machine for 0-CFA:

$$\begin{array}{ll} \hat{\varsigma} \in \hat{\Sigma} & = Exp \times \widehat{Store} \times Cont \\ s \in Storable & = Lam + Cont \\ \kappa \in Cont & = \mathbf{mt} \mid \mathbf{ar}(e,a) \mid \mathbf{fn}(lam,a) \\ a \in Addr & = Exp + Var \end{array}$$

In Haskell:

```
type Σ = (Exp,Store,Kont)
data Storable = Clo Lambda | Cont Kont
type Store = Addr :-> ℙ(Storable)
data Kont = Mt | Ar(Exp,Addr) | Fn(Lambda,Addr)
data Addr = KAddr Exp | BAddr Var
```

$$
\begin{array}{ll}
\langle x, \hat{\sigma}, \kappa\rangle & \longmapsto_{0CFA} \langle v, \hat{\sigma}, \kappa\rangle \text{ where } v \in \hat{\sigma}(x) \\
\langle (e_0 e_1), \hat{\sigma}, \kappa\rangle & \longmapsto_{0CFA} \langle e_0, \hat{\sigma} \sqcup [a \mapsto \{\kappa\}], \mathbf{ar}(e_1, a)\rangle \text{ where } a = (e_0 e_1) \\
\langle v, \hat{\sigma}, \mathbf{ar}(e, a)\rangle & \longmapsto_{0CFA} \langle e, \hat{\sigma}, \mathbf{fn}(v, a)\rangle \\
\langle v, \hat{\sigma}, \mathbf{fn}((\lambda x.e), a)\rangle & \longmapsto_{0CFA} \langle e, \hat{\sigma} \sqcup [x \mapsto \{v\}], \kappa\rangle \text{ where } \kappa \in \hat{\sigma}(a)
\end{array}
$$

In Haskell:

```
step :: Σ -> [Σ]
step (Ref x, σ, κ) =
     [ (Lam lam, σ, κ)
     | Clo(lam) <- Data.Set.toList $! σ!!(BAddr x) ]

step (f :@ e, σ, κ) = [ (f, σ', Ar(e,a')) ]
 where σ' = σ ⊔ [a' ==> s(Cont κ)]
       a' = KAddr (f :@ e)

step (Lam lam, σ, Ar(e, a')) = [ (e, σ, Fn(lam, a')) ]

step (Lam lam, σ, Fn(x :=> e, a))
   = [ (e, σ ⊔ [BAddr x ==> s(Clo(lam))], κ)
     | Cont κ <- Data.Set.toList $! σ!!a ]
```

2.5 Widening to improve complexity

If implemented naïvely, it takes time exponential in the size of the input program to compute the reachable states of the abstracted machines. Consider the size of the state-space for the abstract time-stamped CESK* machine:

$$
\begin{aligned}
&|Exp \times Env \times \widehat{Store} \times Kont \times Time| \\
&= |Exp| \times |Addr|^{|Var|} \times |Storable|^{|Addr|} \times |Kont| \times |Time|.
\end{aligned}
$$

Without simplifying any further, we clearly have an exponential number of abstract states.

To reduce complexity, we can employ widening in the form of Shivers's single-threaded store (Shivers 1991). To use a single threaded store, we have to reconsider the abstract evaluation function itself. Instead of seeing it as a function that returns the set of reachable states, it is a function that returns a set of partial states plus a single globally approximating store, *i.e.*, $aval : Exp \to System$, where:

$$System = \mathscr{P}(Exp \times Env \times Kont \times Time) \times \widehat{Store}.$$

We compute this as a fixed point of a monotonic function, $f : \mathit{System} \to \mathit{System}$:

$$\begin{aligned}
f(C,\hat{\sigma}) &= (C',\hat{\sigma}'') \text{ where} \\
&\quad Q' = \left\{(c',\hat{\sigma}') : c \in C \text{ and } (c,\hat{\sigma}) \longmapsto (c',\hat{\sigma}')\right\} \\
(c_0,\hat{\sigma}_0) &\cong \mathit{inj}(e) \\
&\quad C' = C \cup \left\{c' : (c',_) \in Q'\right\} \cup \{c_0\} \\
&\quad \hat{\sigma}'' = \hat{\sigma} \sqcup \bigsqcup_{(_,\hat{\sigma}') \in Q'} \hat{\sigma}',
\end{aligned}$$

so that $\mathit{aval}(e) = \mathit{lfp}(f)$. The maximum number of iterations of the function f times the cost of each iteration bounds the complexity of the analysis.

2.6 Polynomial complexity for monovariance

It is straightforward to compute the cost of a monovariant (in our framework, a "0CFA-like") analysis with this widening. In a monovariant analysis, environments disappear; the system-space simplifies to:

$$\begin{aligned}
\mathit{System}_0 &= \mathscr{P}(\mathit{Exp} \times \mathit{Cont}) \times \widehat{\mathit{Store}} \\
&\cong (\mathit{Exp} \to \mathscr{P}(\mathit{Cont})) \times (\mathit{Addr} \to \mathscr{P}(\mathit{Storable})).
\end{aligned}$$

If ascended monotonically, one could add one new partial state each time or introduce a new entry into the global store. Thus, the maximum number of monovariant iterations is:

$$|\mathit{Exp}| \times |\mathit{Cont}| + |\mathit{Addr}| \times |\mathit{Storable}|$$

which is polynomial in the size of the program:

$$|\mathit{Exp}| \times \overbrace{(1 + |\mathit{Exp}|^2 + |\mathit{Exp}|^2)}^{|\mathit{Cont}|} + \overbrace{(|\mathit{Var}| + |\mathit{Exp}|)}^{|\mathit{Addr}|} \times \overbrace{(|\mathit{Lam}| + (1 + |\mathit{Exp}|^2 + |\mathit{Exp}|^2))}^{|\mathit{Storable}|}$$

3 Analyzing by-need with Krivine's machine

Even though the abstract machines of the prior section have advantages over traditional CFAs, the approach we took (store-allocated continuations) yields more novel results when applied in a different context: a lazy variant of Krivine's machine. That is, we can construct an abstract interpreter that both analyzes and exploits laziness. Specifically, we present an abstract analog to a lazy and properly tail-recursive variant of Krivine's machine (1985; 2007) derived by Ager, Danvy, and Midtgaard (Ager et al. 2004). The derivation from Ager *et al.*'s machine to the abstract interpreter follows the same outline as that of Section 1: we apply a pointer refinement by store-allocating continuations and carry out approximation by bounding the store.

The by-need variant of Krivine's machine considered here uses the common implementation technique of store-allocating thunks and forced values. When an application is evaluated, a thunk is created that will compute the value of the argument when forced. Evaluating a variable bound to a thunk causes the thunk to be forced, which updates the store to point to the value produced by evaluating the thunk, then produces that value. Otherwise, evaluating a variable bound to a forced value just produces that value.

Storable values include delayed computations (thunks) $\mathbf{d}(e,\rho)$, and computed values $\mathbf{c}(v,\rho)$, which are just tagged closures. There are two continuation constructors: $\mathbf{c_1}(a,\kappa)$ is induced by a variable occurrence whose binding has not yet been forced to a value. The address a is where we want to write the given value when this continuation is invoked. The other: $\mathbf{c_2}(a,\kappa)$ is induced by an application expression, which forces the operator expression to a value. The address a is the address of the argument.

The concrete state-space is defined as follows and the transition relation is defined as:

$$\begin{array}{ll} \varsigma \in \Sigma & = \mathit{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Cont} \\ s \in \mathit{Storable} & = \mathbf{d}(e,\rho) \mid \mathbf{c}(v,\rho) \\ \kappa \in \mathit{Cont} & = \mathbf{mt} \mid \mathbf{c_1}(a,\kappa) \mid \mathbf{c_2}(a,\kappa) \end{array}$$

$$\begin{array}{lll} \langle x,\rho,\sigma,\kappa\rangle & \longmapsto_{LK} & \langle e,\rho',\sigma,\mathbf{c_1}(\rho(x),\kappa)\rangle, \text{ if } \sigma(\rho(x)) = \mathbf{d}(e,\rho') \\ \langle x,\rho,\sigma,\kappa\rangle & \longmapsto_{LK} & \langle v,\rho',\sigma,\kappa\rangle, \text{ if } \sigma(\rho(x)) = \mathbf{c}(v,\rho') \\ \langle (e_0 e_1),\rho,\sigma,\kappa\rangle & \longmapsto_{LK} & \langle e_0,\rho,\sigma[a \mapsto \mathbf{d}(e_1,\rho)],\mathbf{c_2}(a,\kappa)\rangle \text{where } a \notin \mathit{dom}(\sigma) \\ \langle v,\rho,\sigma,\mathbf{c_1}(a,\kappa)\rangle & \longmapsto_{LK} & \langle v,\rho,\sigma[a \mapsto \mathbf{c}(v,\rho)],\kappa\rangle \\ \langle (\lambda x.e),\rho,\sigma,\mathbf{c_2}(a,\kappa)\rangle & \longmapsto_{LK} & \langle e,\rho[x \mapsto a],\sigma,\kappa\rangle \end{array}$$

When the control component is a variable, the machine looks up its stored value, which is either computed or delayed. If delayed, a $\mathbf{c_1}$ continuation is pushed and the frozen expression is put in control. If computed, the value is simply returned. When a value is returned to a $\mathbf{c_1}$ continuation, the store is updated to reflect the computed value. When a value is returned to a $\mathbf{c_2}$ continuation, its body is put in control and the formal parameter is bound to the address of the argument.

We now refactor the machine to use store-allocated continuations; storable values are extended to include continuations:

$$\begin{array}{ll} \varsigma \in \Sigma & = \mathit{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Addr} \\ s \in \mathit{Storable} & = \mathbf{d}(e,\rho) \mid \mathbf{c}(v,\rho) \mid \kappa \\ \kappa \in \mathit{Cont} & = \mathbf{mt} \mid \mathbf{c_1}(a,a) \mid \mathbf{c_2}(a,a). \end{array}$$

It is straightforward to perform a pointer-refinement of the LK machine to store-allocate continuations as done for the CESK machine in Section 1.5 and observe the lazy variant of Krivine's machine and its pointer-refined counterpart (not shown) operate in lock-step:

Lemma 4
$\mathit{eval}_{LK}(e) \simeq \mathit{eval}_{LK^\star}(e)$.

After threading time-stamps through the machine as done in Section 1.7 and defining $\widehat{tick}$ and $\widehat{alloc}$ analogously to the definitions given in Section 1.8, the pointer-refined machine abstracts directly to yield the abstract LK* machine:

$$\begin{array}{lll} \langle x,\rho,\hat{\sigma},\kappa,t\rangle & \longmapsto_{LK^\star_t} & \langle e,\rho',\hat{\sigma} \sqcup [a_0 \mapsto \kappa],\mathbf{c_1}(\rho(x),a_0),u\rangle \text{ if } \hat{\sigma}(\rho(x)) \ni \mathbf{d}(e,\rho') \\ \langle x,\rho,\hat{\sigma},\kappa,t\rangle & \longmapsto_{LK^\star_t} & \langle v,\rho',\hat{\sigma},\kappa,u\rangle \text{ if } \hat{\sigma}(\rho(x)) \ni \mathbf{c}(v,\rho') \\ \langle (e_0 e_1),\rho,\hat{\sigma},\kappa,t\rangle & \longmapsto_{LK^\star_t} & \langle e_0,\rho,\hat{\sigma} \sqcup [a_0 \mapsto \mathbf{d}(e_1,\rho), a_1 \mapsto \kappa],\mathbf{c_2}(c,a_0),u\rangle \\ \langle v,\rho,\hat{\sigma},\mathbf{c_1}(a',c),t\rangle & \longmapsto_{LK^\star_t} & \langle v,\rho',\hat{\sigma} \sqcup [a' \mapsto \mathbf{c}(v,\rho)],\kappa,u\rangle \text{ if } \kappa \in \hat{\sigma}(c) \\ \langle (\lambda x.e),\rho,\hat{\sigma},\mathbf{c_2}(a,c),t\rangle & \longmapsto_{LK^\star_t} & \langle e,\rho'[x \mapsto a],\hat{\sigma},\kappa,u\rangle \text{ if } \kappa \in \hat{\sigma}(c) \end{array}$$

where $a_{0..n} = \widehat{alloc}(\hat{\varsigma})$ and $u = \widehat{tick}(\hat{\varsigma})$.

The abstraction map for this machine is a straightforward structural abstraction similar to that given in Section 1.9 (and hence omitted). The abstracted machine is sound with respect to the LK* machine, and therefore the original LK machine.

Theorem 3 (*Soundness of the Abstract LK* Machine*)

If $\varsigma \longmapsto_{LK} \varsigma'$ and $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$, then there exists an abstract state $\hat{\varsigma}'$, such that $\hat{\varsigma} \longmapsto_{\widehat{LK^\star_t}} \hat{\varsigma}'$ and $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$.

3.1 Optimizing the machine through specialization

Ager *et al.* optimize the LK machine by specializing application transitions. When the operand of an application is a variable, no delayed computation needs to be constructed, thus "avoiding the construction of space-leaky chains of thunks." Likewise, when the operand is a λ-abstraction, "we can store the corresponding closure as a computed value rather than as a delayed computation." Both of these optimizations, which conserve valuable abstract resources, can be added with no trouble:

$$\langle (ex), \rho, \hat{\sigma}, \kappa, t\rangle \longmapsto_{\widehat{LK^\star}} \langle e, \rho, \hat{\sigma} \sqcup [a_0 \mapsto \kappa], \mathbf{c_2}(\rho(x), a_0), u\rangle$$
$$\langle (ev), \rho, \hat{\sigma}, \kappa, t\rangle \longmapsto_{\widehat{LK^\star}} \langle e_0, \rho, \hat{\sigma} \sqcup [a_0 \mapsto \mathbf{c}(v,\rho), a_1 \mapsto \kappa], \mathbf{c_2}(a_0, a_1), u\rangle$$

where $a_{0..n} = \widehat{alloc}(\hat{\varsigma})$ and $u = \widehat{tick}(\hat{\varsigma})$.

3.2 Varying the machine through postponed thunk creation

Ager *et al.* also vary the LK machine by postponing the construction of a delayed computation from the point at which an application is the control string to the point at which the operator has been evaluated and is being applied. The $\mathbf{c_2}$ continuation is modified to hold, rather than the address of a delayed computation, the constituents of the computation itself:

$$\kappa \in Cont = \mathbf{mt} \mid \mathbf{c_1}(a,a) \mid \mathbf{c_2}(e,\rho,a).$$

The transitions for applications and functions are replaced with:

$$\langle (e_0 e_1), \rho, \hat{\sigma}, \kappa, t\rangle \longmapsto_{\widehat{LK'^\star}} \langle e_0, \rho, \hat{\sigma} \sqcup [a_0 \mapsto \kappa], \mathbf{c_2}(e_1, \rho, a_0), u\rangle$$
$$\langle (\lambda x.e), \rho, \hat{\sigma}, \mathbf{c_2}(e', \rho', c), t\rangle \longmapsto_{\widehat{LK'^\star}} \langle e, \rho[x \mapsto a_0], \hat{\sigma} \sqcup [a_0 \mapsto \mathbf{d}(e', \rho')], \kappa, u\rangle \text{ if } \kappa \in \hat{\sigma}(c)$$

where $a_{0..n} = \widehat{alloc}(\hat{\varsigma})$ and $u = \widehat{tick}(\hat{\varsigma})$. This allocates thunks when a function is applied, rather than when the control string is an application.

As Ager *et al.* remark, each of these variants gives rise to an abstract machine. From each of these machines, we are able to systematically derive their abstractions.

4 State and control

We have shown that store-allocated continuations make abstract interpretation of the CESK machine and a lazy variant of Krivine's 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, exceptions and 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 abstract interpreters.

4.1 Conditionals, mutation, and control

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 $E[\texttt{(if [ ] } e_0\ e_1\texttt{)}]$ 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.

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 (Felleisen et al. 2009, page 166). The evaluation context $E[\texttt{(set! } x\ \texttt{[ ])}]$ 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:

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

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.

$$
\begin{array}{ll}
\langle (\texttt{if}\ e_0\ e_1\ e_2), \rho, \hat{\sigma}, \kappa, t\rangle & \longmapsto_{\widehat{CESK}^\star_t} \langle e_0, \rho, \hat{\sigma} \sqcup [a \mapsto \kappa], \mathbf{if}(e_1, e_2, \rho, a), u\rangle \\
\langle \texttt{\#f}, \rho, \hat{\sigma}, \mathbf{if}(e_0, e_1, \rho', c), t\rangle & \longmapsto_{\widehat{CESK}^\star_t} \langle e_1, \rho', \hat{\sigma}, \kappa, u\rangle \text{ if } \kappa \in \hat{\sigma}(c) \\
\langle v, \rho, \hat{\sigma}, \mathbf{if}(e_0, e_1, \rho', c), t\rangle & \longmapsto_{\widehat{CESK}^\star_t} \langle e_0, \rho', \hat{\sigma}, c, u\rangle \text{ if } \kappa \in \hat{\sigma}(c) \text{ and } v \neq \texttt{\#f} \\
\langle (\texttt{set!}\ x\ e), \rho, \hat{\sigma}, \kappa, t\rangle & \longmapsto_{\widehat{CESK}^\star_t} \langle e, \rho, \hat{\sigma} \sqcup [a \mapsto \kappa], \mathbf{set}(\rho(x), a), u\rangle \\
\langle v, \rho, \hat{\sigma}, \mathbf{set}(a', c), t\rangle & \longmapsto_{\widehat{CESK}^\star_t} \langle v', \rho, \hat{\sigma} \sqcup [a' \mapsto v], \kappa, u\rangle \\
 & \quad \text{if } \kappa \in \hat{\sigma}(c) \text{ and } v' \in \hat{\sigma}(a') \\
\langle (\lambda x.e), \rho, \hat{\sigma}, \mathbf{fn}(\texttt{callcc}, \rho', c), t\rangle & \longmapsto_{\widehat{CESK}^\star_t} \langle e, \rho[x \mapsto a], \hat{\sigma} \sqcup [a \mapsto \kappa], \kappa, u\rangle \text{ if } \kappa \in \hat{\sigma}(c) \\
\langle \kappa, \rho, \hat{\sigma}, \mathbf{fn}(\texttt{callcc}, \rho', c), t\rangle & \longmapsto_{\widehat{CESK}^\star_t} \langle \mathbf{fn}(\texttt{callcc}, \rho', c), \rho, \hat{\sigma}, \kappa, u\rangle \\
\langle v, \rho, \hat{\sigma}, \mathbf{fn}(\kappa, \rho', a'), t\rangle & \longmapsto_{\widehat{CESK}^\star_t} \langle v, \rho, \hat{\sigma}, \kappa, u\rangle
\end{array}
$$

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.

4.2 Exceptions and handlers

To analyze exceptional control flow, we extend the CESK machine with a register to hold a stack of exception handlers. This models a reduction semantics in which we have two additional kinds of evaluation contexts:

$$
\begin{array}{l}
E = [\,] \mid (E\,e) \mid (v\,E) \mid (\texttt{catch}\ E\ v) \\
F = [\,] \mid (F\,e) \mid (v\,F) \\
H = [\,] \mid H[F[(\texttt{catch}\ H\ v)]],
\end{array}
$$

and the additional, context-sensitive, notions of reduction:

$$
\begin{array}{c}
(\texttt{catch}\ E[(\texttt{throw}\ v)]\ v') \to (v'\,v) \\
(\texttt{catch}\ v\ v') \to v.
\end{array}
$$

Here, H contexts represent a stack of exception handlers, while F contexts represent a "local" continuation, *i.e.*, the rest of the computation (with respect to the hole) up to an enclosing handler, if any. E contexts represent the entire rest of the computation, including handlers.

The language is extended with expressions for raising and catching exceptions. A new kind of continuation is introduced to represent a stack of handlers. In each frame of the

stack, there is a procedure for handling an exception and a (handler-free) continuation:

$$\begin{array}{rl} e \in \mathit{Exp} & = \ldots \mid \texttt{(throw}\ v\texttt{)} \mid \texttt{(catch}\ e\ \texttt{(}\lambda x.e\texttt{))} \\ \eta \in \mathit{Handl} & = \mathbf{mt} \mid \mathbf{hn}(v, \rho, \kappa, \eta) \end{array}$$

An η continuation represents a stack of exception handler contexts, *i.e.*, $\mathbf{hn}(v', \rho, \kappa, \eta)$ represents $H[F[\texttt{(catch}\ [\,]\ v\texttt{)}]]$, where η represents H, κ represents F, and ρ closes v' to represent v.

The machine includes all of the transitions of the CESK machine extended with a η component; these transitions are omitted for brevity. The additional transitions are:

$$\begin{array}{lll} \langle v, \rho, \sigma, \mathbf{hn}(v', \rho', \kappa, \eta), \mathbf{mt}\rangle & \longmapsto_{\mathit{CESHK}} & \langle v, \rho, \sigma, \eta, \kappa\rangle \\ \langle \texttt{(throw}\ v\texttt{)}, \rho, \sigma, \mathbf{hn}(\texttt{(}\lambda x.e\texttt{)}, \rho', \kappa', \eta), \kappa\rangle & \longmapsto_{\mathit{CESHK}} & \langle e, \rho'[x \mapsto a], \sigma[a \mapsto (v, \rho)], \eta, \kappa'\rangle \\ & & \quad \text{where } a \notin \mathit{dom}(\sigma) \\ \langle \texttt{(catch}\ e\ v\texttt{)}, \rho, \sigma, \eta, \kappa\rangle & \longmapsto_{\mathit{CESHK}} & \langle e, \rho, \sigma, \mathbf{hn}(v, \rho, \kappa, \eta), \mathbf{mt}\rangle \end{array}$$

This presentation is based on a textbook treatment of exceptions and handlers (Felleisen et al. 2009, page 135). To be precise, Felleisen *et al.* present the CHC machine, a substitution-based machine that uses evaluation contexts in place of continuations. Deriving the CESHK machine from it is an easy exercise.

The initial configuration is given by:

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

In the pointer-refined machine, the grammar of handler continuations changes to the following:

$$\eta \in \mathit{Handl} = \mathbf{mt} \mid \mathbf{hn}(v, \rho, a),$$

where a is used to range over addresses pointing to a pair of η and κ continuations. The pointer-refined machine is:

$$\begin{array}{lll} \langle v, \rho, \sigma, \mathbf{hn}(v', \rho', a), \mathbf{mt}\rangle & \longmapsto_{\mathit{CESHK}^\star} & \langle v, \rho, \sigma, \eta, \kappa\rangle \text{ if } (\eta, \kappa) = \sigma(a) \\ \langle \texttt{(throw}\ v\texttt{)}, \rho, \sigma, \mathbf{hn}(\texttt{(}\lambda x.e\texttt{)}, \rho', a), \kappa\rangle & \longmapsto_{\mathit{CESHK}^\star} & \langle e, \rho'[x \mapsto b], \sigma[b \mapsto (v, \rho)], \eta, \kappa'\rangle \\ & & \quad \text{if } (\eta, \kappa') = \sigma(a) \\ \langle \texttt{(catch}\ e\ v\texttt{)}, \rho, \sigma, \eta, \kappa\rangle & \longmapsto_{\mathit{CESHK}^\star} & \langle e, \rho, \sigma[a \mapsto (\eta, \kappa)], \mathbf{hn}(v, \rho, a), \mathbf{mt}\rangle \end{array}$$

After threading time-stamps through the machine as done in Section 1.7, the machine abstracts as expected:

$$\begin{array}{lll} \langle v, \rho, \hat{\sigma}, \mathbf{hn}(v', \rho', a), \mathbf{mt}, t\rangle & \longmapsto_{\widehat{\mathit{CESHK}}^\star_t} & \langle v, \rho, \hat{\sigma}, \eta, \kappa, u\rangle \text{ if } (\eta, \kappa) \in \hat{\sigma}(a) \\ \langle \texttt{(throw}\ v\texttt{)}, \rho, \hat{\sigma}, \mathbf{hn}(\texttt{(}\lambda x.e\texttt{)}, \rho', a), \kappa, t\rangle & \longmapsto_{\widehat{\mathit{CESHK}}^\star_t} & \langle e, \rho'[x \mapsto b], \hat{\sigma} \sqcup [b \mapsto (v, \rho)], \eta, \kappa', u\rangle \\ & & \quad \text{if } (\eta, \kappa') \in \sigma(a) \\ \langle \texttt{(catch}\ e\ v\texttt{)}, \rho, \hat{\sigma}, \eta, \kappa, t\rangle & \longmapsto_{\widehat{\mathit{CESHK}}^\star_t} & \langle e, \rho, \hat{\sigma} \sqcup [a \mapsto (\eta, \kappa)], \mathbf{hn}(v, \rho, a, h), \mathbf{mt}, u\rangle \end{array}$$

5 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 (Morrisett et al. 1995; Felleisen et al. 2009, page 172):

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

This machine iterates over a set of reachable but unvisited "grey" locations $\mathscr{G}$. On each iteration, an element is removed and added to the set of reachable and visited "black" locations $\mathscr{B}$. 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 will vary based on the particular machine being garbage collected, but the definition appropriate for the CESK* machine of Section 1.5 is

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

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 of the machine 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:

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

The CESK* machine incorporates garbage collection by a transition rule that invokes the GC machine as a subroutine to remove garbage from the store. The garbage collection transition introduces nondeterminism to the CESK* machine because it applies to any machine state and thus overlaps with the existing transition rules. The nondeterminism 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 below, only 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.

$$\langle e, \rho, \sigma, \kappa \rangle \longmapsto_{CESK^\star} \langle e, \rho, \{\langle b, \sigma(b)\rangle \mid b \in \mathscr{L}\}, \kappa\rangle$$
$$\text{if } \langle LL_\sigma(e,\rho) \cup LL_\sigma(\kappa), \emptyset, \sigma\rangle \longmapsto\!\!\!\!\rightarrow_{GC} \langle \emptyset, \mathscr{L}, \sigma\rangle$$

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 followed by the GC transition. 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.

6 Abstract stack inspection

In this section, we derive an abstract interpreter for the static analysis of a higher-order language with stack inspection. Following the outline of Section 1 and 3, we start from the tail-recursive CM machine of Clements and Felleisen (2004), perform a pointer refinement on continuations, then abstract the semantics by a parameterized bounding of the store.

6.1 The λ_{sec}-calculus and stack-inspection

The λ_{sec}-calculus of Pottier et al. (2005) is a call-by-value λ-calculus model of higher-order stack inspection. We present the language as given by Clements and Felleisen (2004).

All code is statically annotated with a given set of permissions R, chosen from a fixed set $\mathscr{P}$. A computation whose source code was statically annotated with a permission may *enable* that permission for the dynamic extent of a subcomputation. The subcomputation is privileged so long as it is annotated with the same permission, and every intervening procedure call has likewise been annotated with the privilege.

$$e \in Exp = \ldots \mid \texttt{fail} \mid (\texttt{grant}\ R\ e) \mid (\texttt{test}\ R\ e\ e) \mid (\texttt{frame}\ R\ e)$$

A `fail` expression signals an exception if evaluated; by convention it is used to signal a stack-inspection failure. A `(frame` R e`)` evaluates e as the principal R, representing the permissions conferred on e given its origin. A `(grant` R e`)` expression evaluates as e but with the permissions extended with R enabled. A `(test` R e_0 e_1`)` expression evaluates to e_0 if R is enabled and e_1 otherwise.

A trusted annotator consumes a program and the set of permissions it will operate under and inserts frame expressions around each λ-body and intersects all grant expressions with this set of permissions. We assume all programs have been properly annotated.

Stack inspection can be understood in terms of an *OK* predicate on an evaluation contexts and permissions. The predicate determines whether the given permissions are enabled for a subexpression in the hole of the context. The *OK* predicate holds whenever the context can be traversed from the hole outwards and, for each permission, find an enabling grant context without first finding a denying frame context.

6.2 The CM machine

The continuation marks (CM) machine of Clements and Felleisen (2004) is a properly tail-recursive extended CESK machine for interpreting higher-order languages with stack-inspection. In the CM machine, continuations are annotated with *marks* (Clements et al. 2001), which, for the purposes of stack-inspection, are finite maps from permissions to $\{\text{deny}, \text{grant}\}$:

$$\kappa \quad = \quad \mathbf{mt}^m \mid \mathbf{ar}^m(e,\rho,\kappa) \mid \mathbf{fn}^m(v,\rho,\kappa).$$

We write $\kappa[R \mapsto c]$ to denote updating the marks on κ to $m[R \mapsto c]$.

The CM machine is defined as follows, where transitions that are straightforward adaptations of the corresponding CESK* transitions to incorporate continuation marks are omitted:

$$\begin{aligned}
\langle \texttt{fail}, \rho, \sigma, \kappa\rangle &\longmapsto_{CM} \langle \texttt{fail}, \rho, \sigma, \mathbf{mt}^{\emptyset}\rangle \\
\langle \texttt{(frame}\, R\, e\texttt{)}, \rho, \sigma, \kappa\rangle &\longmapsto_{CM} \langle e, \rho, \sigma, \kappa[\overline{R} \mapsto \text{deny}]\rangle \\
\langle \texttt{(grant}\, R\, e\texttt{)}, \rho, \sigma, \kappa\rangle &\longmapsto_{CM} \langle e, \rho, \sigma, \kappa[R \mapsto \text{grant}]\rangle \\
\langle \texttt{(test}\, R\, e_0\, e_1\texttt{)}, \rho, \sigma, \kappa\rangle &\longmapsto_{CM} \begin{cases} \langle e_0, \rho, \sigma, \kappa\rangle & \text{if } OK(R,\kappa), \\ \langle e_1, \rho, \sigma, \kappa\rangle & \text{otherwise} \end{cases}
\end{aligned}$$

It relies on the *OK* predicate to determine whether the permissions in R are enabled. The *OK* predicate performs the traversal of the context (represented as a continuation) using marks to determine which permissions have been granted or denied:

$$\begin{aligned}
OK(\emptyset, \kappa) & \\
OK(R, \mathbf{mt}^m) \quad &\Longleftrightarrow \quad (R \cap m^{-1}(\text{deny}) = \emptyset) \\
\left.\begin{array}{r} OK(R, \mathbf{fn}^m(v,\rho,\kappa)) \\ OK(R, \mathbf{ar}^m(e,\rho,\kappa)) \end{array}\right\} \quad &\Longleftrightarrow \quad (R \cap m^{-1}(\text{deny}) = \emptyset) \wedge OK(R \setminus m^{-1}(\text{grant}), \kappa)
\end{aligned}$$

The semantics of a program is given by the set of reachable states from an initial machine configuration:

$$inj_{CM}(e) = \langle e, \emptyset, \emptyset, \mathbf{mt}^{\emptyset}\rangle.$$

6.3 The abstract CM⋆ machine

Store-allocating continuations, time-stamping, and bounding the store yields the transition system given below. It is worth noting that continuation marks are updated, not joined, in the abstract transition system, just as in the concrete.

$$
\begin{array}{ll}
\langle \texttt{fail}, \rho, \hat{\sigma}, \kappa \rangle & \longmapsto_{\widehat{CM}} \langle \texttt{fail}, \rho, \hat{\sigma}, \mathbf{mt}^{\emptyset} \rangle \\
\langle (\texttt{frame}\ R\ e), \rho, \hat{\sigma}, \kappa \rangle & \longmapsto_{\widehat{CM}} \langle e, \rho, \hat{\sigma}, \kappa[\overline{R} \mapsto \text{deny}] \rangle \\
\langle (\texttt{grant}\ R\ e), \rho, \hat{\sigma}, \kappa \rangle & \longmapsto_{\widehat{CM}} \langle e, \rho, \hat{\sigma}, \kappa[R \mapsto \text{grant}] \rangle \\
\langle (\texttt{test}\ R\ e_0\ e_1), \rho, \hat{\sigma}, \kappa \rangle & \longmapsto_{\widehat{CM}} \begin{cases} \langle e_0, \rho, \hat{\sigma}, \kappa \rangle & \text{if } \widehat{OK^\star}(R, \kappa, \hat{\sigma}), \\ \langle e_1, \rho, \hat{\sigma}, \kappa \rangle & \text{otherwise.} \end{cases}
\end{array}
$$

The $\widehat{OK^\star}$ predicate approximates the pointer refinement of its concrete counterpart OK, which can be understood as tracing a path through the store corresponding to traversing the continuation. The abstract predicate holds whenever there exists such a path in the abstract store that would satisfy the concrete predicate:

$$
\begin{array}{rcl}
\widehat{OK^\star}(\emptyset, \kappa, \hat{\sigma}) & & \\
\widehat{OK^\star}(R, \mathbf{mt}^m, \hat{\sigma}) & \iff & (R \cap m^{-1}(\text{deny}) = \emptyset) \\
\left.\begin{array}{l} \widehat{OK^\star}(R, \mathbf{fn}^m(v, \rho, a), \hat{\sigma}) \\ \widehat{OK^\star}(R, \mathbf{ar}^m(e, \rho, a), \hat{\sigma}) \end{array}\right\} & \iff & \begin{array}{l} (R \cap m^{-1}(\text{deny}) = \emptyset) \wedge \widehat{OK^\star}(R \setminus m^{-1}(\text{grant}), \kappa, \hat{\sigma}) \\ \text{where } \kappa \in \hat{\sigma}(a) \end{array}
\end{array}
$$

Consequently, in analyzing $(\texttt{test}\ R\ e_0\ e_1)$, e_0 is reachable only when the analysis can prove the $OK^\star$ predicate holds on some path through the abstract store.

It is straightforward to define a structural abstraction map and verify the abstract CM⋆ machine is a sound approximation of its concrete counterpart:

Theorem 4 (*Soundness of the Abstract CM⋆ Machine*)

If $\varsigma \longmapsto_{CM} \varsigma'$ and $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$, then there exists an abstract state $\hat{\varsigma}'$, such that $\hat{\varsigma} \longmapsto_{\widehat{CM^\star_t}} \hat{\varsigma}'$ and $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$.

7 Pushdown abstractions

Pushdown analysis is an alternative paradigm for the analysis of programs in which the run-time program stack is precisely modeled with the stack of a pushdown system. Consequently, a pushdown analysis can exactly match control flow transfers from calls to returns, from throws to handlers, and from breaks to labels. This in contrast with the traditional approaches of finite-state abstractions we have considered so far, which necessarily model the control stack with finite bounds.

Although pushdown abstractions have been well-known in the setting of first-order languages (Bouajjani et al. 1997; Reps 1998; Kodumal and Aiken 2004), they have eluded extension to a higher-order setting until the recent work of Vardoulakis and Shivers (2011).

In this section, we show that pushdown analysis has a natural expression as an abstraction of a classical abstract machine. We revisit our recipe for abstracting the CESK machine and demonstrate that by store-allocating bindings but *not* continuations, a computable

pushdown model is obtained. The pushdown model more closely resembles its concrete counterpart, hence establishing soundness is even easier. But since the resulting state-space is potentially infinite, decidability becomes less straightforward. We show that the resulting abstract machine is equivalent to a pushdown automata, making it clear that reachability of machine states is a decidable property.

7.1 The abstract pushdown CESK* machine

We have seen that store-allocating bindings and continuations is a powerful technique for transforming abstract machines in to their computable, finite-state approximations. The key to obtaining a pushdown model is to simply replay the same steps but to keep continuations on the control stack rather than moving them to the store. In other words, starting from the CESK machine, which puts bindings in the store, all that is needed for a pushdown analysis is to bound the store.

$$\begin{aligned} \varsigma \in \Sigma &= Exp \times Env \times Store \times Cont \\ \rho \in Env &= Var \to_{\text{fin}} Addr \\ \sigma \in Store &= Addr \to_{\text{fin}} Storable \\ s \in Storable &= Lam \times Env \\ a, b, c \in Addr &\quad \text{an infinite set.} \end{aligned}$$

The machine is defined as:

$$\begin{aligned} \langle x, \rho, \hat{\sigma}, \kappa\rangle &\longmapsto_{\widehat{CESK}} \langle v, \rho', \hat{\sigma}, \kappa\rangle \text{ where } (v, \rho') \in \hat{\sigma}(\rho(x)) \\ \langle (e_0 e_1), \rho, \hat{\sigma}, \kappa\rangle &\longmapsto_{\widehat{CESK}} \langle e_0, \rho, \hat{\sigma}, \mathbf{ar}(e_1, \rho, \kappa)\rangle \\ \langle v, \rho, \hat{\sigma}, \mathbf{ar}(e, \rho', \kappa)\rangle &\longmapsto_{\widehat{CESK}} \langle e, \rho', \hat{\sigma}, \mathbf{fn}(v, \rho, \kappa)\rangle \\ \langle v, \rho, \hat{\sigma}, \mathbf{fn}((\lambda x.e), \rho', \kappa)\rangle &\longmapsto_{\widehat{CESK}} \langle e, \rho'[x \mapsto a], \hat{\sigma} \sqcup [a \mapsto \{(v, \rho)\}], \kappa\rangle, \\ &\qquad \text{where } a = \widehat{alloc}(\hat{\varsigma}). \end{aligned}$$

The abstract pushdown CESK machine makes a nondeterministic choice when dereferencing a variable, however it is completely deterministic in its choice of continuations since the control stack is modeled as a stack just as in the concrete CESK machine. Since the stack has no bound, we no longer have a finite-state space, but as we will see it is still possible to decide whether a given machine state is reachable from the initial configuration.

7.2 Soundness and computability

Theorem 5 (*Soundness of the Abstract Pushdown CESK Machine*)

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

The proof follows the same structure as that of theorem 2, and is in fact simplified since the continuations frames of the machines exactly correspond, eliminating the need to consider the nondeterministic choice of a continuation residing at a store location.

The more interesting aspect of the pushdown abstraction is decidability. Notice that since the stack has a recursive, unbounded structure, the state-space of the machine is potentially infinite so deciding reachability by enumerating the reachable states will no longer suffice.

Theorem 6 (*Decidability of the Abstract Pushdown CESK* Machine*)

$\hat{\varsigma} \in \widehat{aval}_{\widehat{CESK}}(e)$ is decidable.

Proof
Observe that the control string, environment, and store components of a machine state are drawn from finite sets. Continuations may be represented isomorphically as a list of stack frames:

$$\kappa = [f, \ldots] \qquad\qquad f = \mathbf{ar}(e, \rho) \mid \mathbf{fn}(v, \rho)$$

Furthermore, stack frames are drawn from a finite set since expressions and environments are finite. But now observe that the abstract pushdown CESK machine is a pushdown automata: states of the PDA are CES triples from the CESK machine; the PDA stack alphabet is the alphabet of stack frames; and PDA transitions easily encode machine transitions by mapping from a CES triple and stack frame to a new CES triple and pushing/popping the stack appropriately. □

8 Related work

The study of abstract machines for the λ-calculus began with the SECD machine of Landin (1964), the systematic construction of machines from semantics with the definitional interpreters of Reynolds (1972), the theory of abstract interpretation with the POPL papers of Cousot and Cousot (1977, 1979), and static analysis of the λ-calculus with the coupling of abstract machines and abstract interpretation by Jones (1981). 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 (2008, 2009). We strengthen the connection by demonstrating a general technique for abstracting abstract machines.

8.1 Abstract interpretation of abstract machines

The approximation of abstract machine states for the analysis of higher-order languages goes back to Jones (1981), 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.

Ashley and Dybvig (1998) use a non-standard abstract machine as the basis for their concrete semantics. The machine is a CES machine; continuations in the machine are eliminated by transforming programs into explicit continuation-passing style. The machine also collects a *cache*, which maps execution contexts (roughly time-stamps in our setting) to a store describing that context. To abstract, the cache is restricted to a finite function, which is ensured by allocating from a finite set of addresses just as we have done.

Midtgaard and Jensen (2008) present a 0CFA for a CPS λ-calculus language. The approach is based on Cousot-style calculational abstract interpretation (Cousot 1999), applied to a functional language. Like the present work, Midtgaard and Jensen start with an "off-the-shelf" abstract machine for the concrete semantics—in this case, the CE machine of Flanagan et al. (1993)—and employ a reachable-states model. They then compose well-known Galois connections to reveal a 0CFA with reachability in the style of Ayers (1993).[3] The CE machine is not sufficient to interpret direct-style programs, so the analysis is specialized to programs in continuation-passing style. Later work by Midtgaard and Jensen (2009) went on to present a similar calculational abstract interpretation treatment of a monomorphic CFA for an ANF λ-calculus. The concrete semantics are based on reachable states of the C_aEK machine (Flanagan et al. 1993). The abstract semantics approximate the control stack component of the machine by its top element, which is similar to the labeled machine abstraction given in Section 2 when $k = 0$.

Although our approach is not calculational like Midtgaard and Jensen's, it continues in their tradition by applying abstract interpretation to off-the-shelf tail-recursive machines. We extend the application to direct-style machines for a *k*-CFA-like abstraction that handles tail calls, laziness, state, exceptions, first-class continuations, and stack inspection. We have extended *return flow analysis* to a completely direct style (no ANF or CPS needed) within a framework that accounts for polyvariance.

Harrison (1989) gives an abstract interpretation for a higher-order language with control and state for the purposes of automatic parallelization. Harrison maps Scheme programs into an imperative intermediate language, which is interpreted on a novel abstract machine. The machine uses a procedure string approach similar to that given in Section 2 in that the store is addressed by procedure strings. Harrison's first machine employs higher-order values to represent functions and continuations and he notes, "the straightforward abstraction of this semantics leads to abstract domains containing higher-order objects (functions) over reflexive domains, whereas our purpose requires a more concrete compile-time representation of the values assumed by variables. We therefore modify the semantics such that its abstraction results in domains which are both finite and non-reflexive." Because of the reflexivity of denotable values, a direct abstraction is not possible, so he performs closure conversion on the (representation of) the semantic function. Harrison then abstracts the machine by bounding the procedure string space (and hence the store) via an abstraction he calls stack configurations, which is represented by a finite set of members, each of which describes an infinite set of procedure strings.

To prove that Harrison's abstract interpreter is correct he argues that the machine interpreting the translation of a program in the intermediate language corresponds to interpreting the program as written in the standard semantics—in this case, the denotational semantics of R^3RS. On the other hand, our approach relies on well known machines with well known relations to calculi, reduction semantics, and other machines (Felleisen 1987; Danvy 2006). These connections, coupled with the strong similarities between our con-

[3] Ayers derived an abstract interpreter by transforming (the representation of) a denotational continuation semantics of Scheme into a state transition system (an abstract machine), which he then approximated using Galois connections (Ayers 1993).

crete and abstract machines, result in minimal proof obligations in comparison. Moreover, programs are analyzed in direct-style under our approach.

8.2 Abstract interpretation of lazy languages

Jones has analyzed non-strict functional languages (Jones 1981; Jones and Andersen 2007), but that work has only focused on the by-name aspect of laziness and does not address memoization as done here. Sestoft (1991) examines flow analysis for lazy languages and uses abstract machines to prove soundness. In particular, Sestoft presents a lazy variant of Krivine's machine similar to that given in Section 3 and proves analysis is sound with respect to the machine. Likewise, Sestoft uses Landin's SECD machine as the operational basis for proving globalization optimizations correct. Sestoft's work differs from ours in that analysis is developed separately from the abstract machines, whereas we derive abstract interpreters directly from machine definitions. Faxén (1995) uses a type-based flow analysis approach to analyzing a functional language with explicit thunks and evals, which is intended as the intermediate language for a compiler of a lazy language. In contrast, our approach makes no assumptions about the typing discipline and analyzes source code directly.

8.3 Realistic language features and garbage collection

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 (Shivers 1991). 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 (2006) introduced the idea of using abstract garbage collection to improve precision and efficiency in flow analysis. 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.

8.4 Static stack inspection

Most work on the static verification of stack inspection has focused on type-based approaches. Skalka and Smith (2000) present a type system for static enforcement of stack-inspection. Pottier et al. (2005) present type systems for enforcing stack-inspection developed via a static correspondence to the dynamic notion of security-passing style. Skalka et al. (2008) present type and effect systems that use linear temporal logic to express regular properties of program traces and show how to statically enforce both stack- and history-based security mechanisms. Our approach, in contrast, is not typed-based and focuses only on stack-inspection, although it seems plausible the approach of Section 6 extends to the more general history-based mechanisms.

9 Conclusions and perspective

We have demonstrated the utility of store-allocated continuations by deriving novel abstract interpretations of the CEK, a lazy variant of Krivine's, and the stack-inspecting CM machines. 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. Our technique allows concrete implementation technology to be mapped straightforwardly into that of static analysis, which we demonstrated by incorporating abstract garbage collection and optimizations to avoid abstract space leaks, both of which are based on existing accounts of concrete GC and space efficiency. Moreover, the abstract interpreters properly model tail-calls by virtue of their concrete counterparts being properly tail-call optimizing. Finally, our technique uniformly scales up to both richer language features and richer analyses. To support the first claim, we extended the abstract CESK machine to analyze conditionals, first-class control, exception handling, and state. To support the second, we have shown how to adapt the CESK machine for a pushdown analysis. We speculate that store-allocating bindings and continuations is sufficient for a straightforward abstraction of most existing machines.

Acknowledgments

We thank Olivier Danvy, Matthias Felleisen, Jan Midtgaard, Sam Tobin-Hochstadt, and Mitchell Wand for discussions and suggestions. We also thank the anonymous reviewers for their close reading and helpful critiques; their comments have improved this work. We are grateful to Olivier Danvy and Jan Midtgaard for writing a lucid technical perspective on this work for *Communications of the ACM*.

This material is based upon work supported by the National Science Foundation under Grant No. 1035658. The first author was supported by the National Science Foundation under Grant No. 0937060 to the Computing Research Association for the CIFellow Project.

References

Ager, M. S., O. Danvy, and J. Midtgaard (2004, June). A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters 90*(5), 223–232.

Ashley, J. M. and R. K. Dybvig (1998). A practical and flexible flow analysis for higher-order languages. *ACM TOPLAS 20*(4), 845–868.

Ayers, A. E. (1993). *Abstract analysis and optimization of Scheme*. Ph. D. thesis, Massachusetts Institute of Technology.

Bouajjani, A., J. Esparza, and O. Maler (1997). Reachability analysis of pushdown automata: Application to Model-Checking. In *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory*, pp. 135–150.

Clements, J. and M. Felleisen (2004, November). A tail-recursive machine with stack inspection. *ACM Trans. Program. Lang. Syst. 26*(6), 1029–1052.

Clements, J., M. Flatt, and M. Felleisen (2001). Modeling an algebraic stepper. In *ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems*, pp. 320–334.

Cousot, P. (1999). The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen (Eds.), *Calculational System Design*.

Cousot, P. and R. Cousot (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pp. 238–252.

Cousot, P. and R. Cousot (1979). Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pp. 269–282.

Danvy, O. (2006, October). *An Analytical Approach to Program as Data Objects*. DSc thesis, Department of Computer Science, Aarhus University.

Earl, C., M. Might, and D. Van Horn (2010). Pushdown Control-Flow analysis of Higher-Order programs. In *Workshop on Scheme and Functional Programming*.

Faxén, K. (1995). Optimizing lazy functional programs using flow inference. In *Static Analysis*, pp. 136–153.

Felleisen, M. (1987). *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. Ph. D. thesis, Indiana University.

Felleisen, M., R. B. Findler, and M. Flatt (2009, August). *Semantics Engineering with PLT Redex*.

Felleisen, M. and D. P. Friedman (1986a, August). Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*.

Felleisen, M. and D. P. Friedman (1986b, August). Control operators, the SECD-machine, and the Lambda-Calculus. In *Proc. of the IFIP TC 2/WG2. 2 Working Conf. on Formal Description of Programming Concepts Part III*, pp. 193–219.

Felleisen, M. and D. P. Friedman (1987). A calculus for assignments in higher-order languages. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 314–325. POPL'87.

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

Harrison, W. L. (1989, October). The interprocedural analysis and automatic parallelization of scheme programs. *LISP and Symbolic Computation 2*(3), 179–396.

Jones, N. and N. Andersen (2007, May). Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science 375*(1-3), 120–136.

Jones, N. D. (1981). Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pp. 114–128.

Jones, N. D. and S. S. Muchnick (1982). 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, pp. 66–74.

Kodumal, J. and A. Aiken (2004, June). The set constraint/CFL reachability connection in practice. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pp. 207–218.

Krivine, J.-L. (1985). Un interpréteur du lambda-calcul.

Krivine, J.-L. (2007, September). A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation 20*(3), 199–207.

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

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

Midtgaard, J. (2011). Control-flow analysis of functional programs. *ACM Computing Surveys*. To appear.

Midtgaard, J. and T. Jensen (2008). A calculational approach to Control-Flow analysis by abstract interpretation. In M. Alpuente and G. Vidal (Eds.), *SAS*, Volume 5079 of *LNCS*, pp. 347–362.

Midtgaard, J. and T. P. Jensen (2009). 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*, pp. 287–298.

Might, M. and O. Shivers (2006). Improving flow analyses via Gamma-CFA: Abstract garbage collection and counting. In *ICFP '06: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pp. 13–25.

Morrisett, G., M. Felleisen, and R. Harper (1995). Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pp. 66–77.

Nielson, F., H. R. Nielson, and C. Hankin (1999). *Principles of Program Analysis*.

Pottier, F., C. Skalka, and S. Smith (2005, March). A systematic approach to static access control. *ACM Trans. Program. Lang. Syst. 27*(2), 344–382.

Reps, T. (1998, December). Program analysis via graph reachability. *Information and Software Technology 40*(11-12), 701–726.

Reynolds, J. C. (1972). Definitional interpreters for higher-order programming languages. In *ACM 1972: Proceedings of the ACM Annual Conference*, pp. 717–740.

Sestoft, P. (1991, October). *Analysis and efficient implementation of functional programs*. Ph. D. thesis, University of Copenhagen.

Shao, Z. and A. W. Appel (1994). Space-efficient closure representations. In *LFP '94: Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pp. 150–161.

Sharir, M. and A. Pnueli (1981). *Two approaches to interprocedural data flow analysis*, Chapter 7, pp. 189–234.

Shivers, O. G. (1991). *Control-Flow Analysis of Higher-Order Languages*. Ph. D. thesis, Carnegie Mellon University.

Skalka, C. and S. Smith (2000, September). Static enforcement of security with types. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, pp. 34–45.

Skalka, C., S. Smith, and D. Van Horn (2008). Types and trace effects of higher order programs. *Journal of Functional Programming 18*(02), 179–249.

Vardoulakis, D. and O. Shivers (2011, May). CFA2: a Context-Free approach to Control-Flow analysis. *Logical Methods in Computer Science 7*(2).

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