Optimizing
Abstract Abstract Machines

J. Ian Johnson,
Nicholas Labich, David Van Horn
{ianj,labichn,dvanhorn}@ccs.neu.edu
Northeastern University
Boston, MA, USA

Matt Might
matt@might.net
University of Utah
Salt Lake City, UT, USA
Abstract
abstract machines?
Abstract
We describe a derivational approach to abstract interpretation that yields novel and transparently sound static analyses when applied to the abstract machines of abstract machines. To do this, we formally define and support our claims, we transform the CEK machine to a call-by-value λ-calculus machine, and the stack-inspecting CM machine of Clements and Felleisen into abstract interpretations of themselves. The resulting analyses exhibit temporal-ordering of program events, predict return-flow and stack-inspection behavior, and approximate the flow and evaluation of by-used parameters. For all of these machines, we find that a set of well-known concrete machine refactorings, plus a technique we call 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.

1. Introduction
Abstract machines such as the CEK and Krivine’s machines are first-pass state transition systems that represent the core of a real language implementation. Semantics-based program analysis, on the other hand, is concerned with safely approximating intentional properties of such a machine as it runs a program. It seems natural then to want to systematically derive analyses from an abstract machine. With no recursive structure in the state-space, an abstraction of the program does not depend on the stack-implementation structure of the machine. For all of these machines, we show that abstracting well-known machines, our technique delivers static analyses that can reason about by-used evaluation, higher-order functions, tail calls, side effects, stack-structures, exceptions and first-class continuations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

We demonstrate that the technique of refactoring a machine with store-allocated continuations allows a direct structural abstraction by bounding the machine’s store. Thus, we are able to convert semantic techniques used to model language features into static analysis techniques for reasoning about the behavior of these very same features. By abstracting well-known machines, our technique delivers static analyzers that can reason about by-used evaluation, higher-order functions, tail calls, side effects, stack-structures, exceptions and first-class 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 complete static analysis.

## Keywords
abstract machines, abstract interpretation

Abstracting Abstract Machines

David Van Horn
Northeastern University
dvanhorn@ccs.neu.edu

Matthew Might
University of Utah
might@cs.utah.edu

1 A structural abstraction distributes component-, point-, and member-wise.
Abstracting Abstract Machines

Abstract

We describe a derivational approach to abstract interpretation that yields novel and transparently sound static analyses when applied to well-known abstract machines. To obtain the abstract machine, we develop an operational semantics for the abstract machine. We then demonstrate that the abstract machine becomes eligible for conversion into an abstract interpretation of itself. The resulting analyses are parameterized, sound and computable static analyses. We then demonstrate that the time-stamped machine abstracts directly into a parameterized, sound and computable static analysis.

1. Introduction

Abstract machines such as the CKE, Krivine's machine and Krivine's machine are first-order state transition systems that represent the core of real language implementations. Semantics-based program analysis, on the other hand, is concerned with safely approximating intensional properties of such a machine as it runs a program. A structural abstraction distributes component-, point-, and member-wise abstractions of components of the machine's state-space into another that computes its finite approximation. 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 abstractions directly into a parameterized, sound and computable static analysis.
Semantics \rightarrow Analysis
Goal: Directly implementable math

Competitive performance, *minimal effort, simple proofs*

same ballpark, nosebleed seats
Goal: Directly implementable math

Competitive performance,*

Slogan:

Engineering tricks as semantics refactorings

same ballpark, nosebleed seats
An ideal static analysis has 

Performance
An ideal static analysis has

Soundness
Maintainability
Precision
Performance
An ideal static analysis has

Soundness
Maintainability
Precision
Performance

[Van Horn and Might ICFP 2010]
(AAM)
An ideal static analysis has

- Soundness
- Maintainability
- Precision
- Performance

[Van Horn and Might ICFP 2010] (AAM)

[Johnson, et al. ICFP 2013] (OAAM)
OAAM outline
OAAM outline

- Store-allocate values
- Frontier-based semantics
- Lazy non-determinism
- Abstract compilation
- Locally log-based store-deltas
- Store-counting
- Mutable frontier and store
OAAM outline

- Decrease state space
  - Store-allocate values
  - Frontier-based semantics
  - Lazy non-determinism
  - Abstract compilation
    - Locally log-based store-deltas
    - Store-counting
    - Mutable frontier and store
OAAM outline

Decrease state space

- Store-allocate values
- Frontier-based semantics
- Lazy non-determinism
- Abstract compilation
- Locally log-based store-deltas
- Store-counting
- Mutable frontier and store

Explore faster
OAAM outline

Decrease state space

- Store-allocate values
- Frontier-based semantics

1. Lazy non-determinism
2. Abstract compilation
3. Locally log-based store-deltas

- Store-counting
- Mutable frontier and store

Explore faster
First: recap of AAM
Start with CESK machine

Control

\[ \langle e, \rho, \sigma, \kappa \rangle \]
Start with CESK machine

\[ \langle e, \rho, \sigma, \kappa \rangle \]

Environment
Start with CESK machine

Store

\[ \langle e, \rho, \sigma, \kappa \rangle \]
Start with CESK machine

\[ \langle e, \rho, \sigma, \kappa \rangle \]

Kontinuation
\[ \rho \in \text{Env} = \text{Var} \rightarrow \text{Addr} \]
\[ \kappa \in \text{Kont} ::= [\cdot] | \phi : \kappa \]
\[ \sigma \in \text{Store} = \text{Addr} \rightarrow (\text{Value} \times \text{Env}) \]
\[ \langle x, \rho, \sigma, \kappa \rangle \mapsto \langle v, \rho', \sigma, \kappa \rangle \]
\[ \text{if } (v, \rho') = \sigma(\rho(x)) \]
\[ \langle (e_0 e_1), \rho, \sigma, \kappa \rangle \mapsto \langle e_0, \rho, \sigma, \text{ar}(e_1, \rho) : \kappa \rangle \]
\[ \langle v, \rho, \sigma, \text{ar}(e, \rho) : \kappa \rangle \mapsto \langle e, \rho, \sigma, \text{fn}(v, \rho) : \kappa \rangle \]
\[ \langle v, \rho, \sigma, \text{fn}(\lambda x. e, \rho') : \kappa \rangle \mapsto \langle e, \rho'', \sigma', \kappa \rangle \]

where \( \rho'' = \rho'[x \mapsto a] \)
\[ \sigma' = \sigma[a \mapsto (v, \rho)] \]
\[ a \text{ fresh} \]
\(\rho \in \text{Env} = \text{Var} \to \text{Addr}\)

\(\kappa \in \text{Kont} ::= [] | \phi : a\)

\(\sigma \in \text{Store} = \text{Addr} \to (\text{Value} \times \text{Env} + \text{Kont})\)

\[
\langle x, \rho, \sigma, \kappa \rangle \mapsto \langle v, \rho', \sigma, \kappa \rangle
\]

If \((v, \rho') = \sigma(\rho(x))\)

\[
\langle (e_0 e_1), \rho, \sigma, \kappa \rangle \mapsto \langle e_0, \rho, \sigma', \text{ar}(e_1, \rho): a \rangle \quad \sigma' = \sigma[a \mapsto \kappa]
\]

\[
\langle v, \rho, \sigma, \text{ar}(e, \rho): b \rangle \mapsto \langle e, \rho, \sigma, \text{fn}(v, \rho): b \rangle
\]

\[
\langle v, \rho, \sigma, \text{fn}(\lambda x.e, \rho') : b \rangle \mapsto \langle e, \rho'', \sigma', \kappa \rangle \quad \kappa = \sigma(b)
\]

where \(\rho'' = \rho'[x \mapsto a]\)

\(\sigma' = \sigma[a \mapsto (v, \rho)]\)

\(a\) fresh
\[ \rho \in \text{Env} = \text{Var} \rightarrow \text{Addr} \]
\[ \kappa \in \text{Kont} ::= [] | \varphi : a \]
\[ \sigma \in \text{Store} = \text{Addr} \rightarrow \rho(\text{Value} \times \text{Env} + \text{Kont}) \]

\[ \langle x, \rho, \sigma, \kappa \rangle \mapsto \langle v, \rho', \sigma, \kappa \rangle \]

if \((v, \rho') \in \sigma(\rho(x))\)

\[ \langle (e_0,e_1), \rho, \sigma, \kappa \rangle \mapsto \langle e_0, \rho, \sigma', \text{ar}(e_1, \rho) : a \rangle \]
\[ \sigma' = \sigma \sqcup [a \mapsto \{v, \rho\}] \]

\[ \langle v, \rho, \sigma, \text{ar}(e, \rho) : b \rangle \mapsto \langle e, \rho, \sigma, \text{fn}(v, \rho) : b \rangle \]

\[ \langle v, \rho, \sigma, \text{fn}(\lambda x.e, \rho') : b \rangle \mapsto \langle e, \rho'', \sigma', \kappa \rangle \]

where \(\rho'' = \rho' \upharpoonright x \mapsto a\)

\[ \sigma' = \sigma \sqcup [a \mapsto \{(v, \rho)\}] \]

\[ a = \text{alloc}(\varsigma) \]
\[ \langle x, \rho, \sigma, k \rangle \mapsto \langle v, \rho', \sigma, k \rangle \]
Finite reduction relation = graph
1. Lazy non-determinism
Finite reduction relation = graph

1. Lazy non-determinism
2. Abstract compilation
1. Lazy non-determinism
2. Abstract compilation
3. Store deltas
1. Lazy non-determinism
2. Abstract compilation
3. Store deltas
\( \rho \in \text{Env} = \text{Var} \rightarrow \text{Addr} \)

\( \kappa \in \text{Kont} ::= [] | \varphi : a \)

\( \sigma \in \text{Store} = \text{Addr} \rightarrow \wp(\text{Value} \times \text{Env} + \text{Kont}) \)

\[ \langle x, \rho, \sigma, \kappa \rangle \mapsto \langle v, \rho', \sigma, \kappa \rangle \]

**if** \( (v, \rho') \in \sigma(\rho(x)) \)

\[ \langle (e_0 \ e_1), \rho, \sigma, \kappa \rangle \mapsto \langle e_0, \rho, \sigma', \text{ar}(e_1, \rho):a \rangle \sigma' = \sigma \sqcup [a \mapsto \{ \kappa \}] \]

\[ \langle v, \rho, \sigma, \text{ar}(e, \rho):b \rangle \mapsto \langle e, \rho, \sigma, \text{fn}(v, \rho):b \rangle \]

\[ \langle v, \rho, \sigma, \text{fn}(\lambda x. e, \rho') : b \rangle \mapsto \langle e, \rho'', \sigma', \kappa \rangle \kappa \in \sigma(b) \]

where \( \rho'' = \rho'[x \mapsto a] \)

\[ \sigma' = \sigma \sqcup [a \mapsto \{(v, \rho)\}] \]

\( a = \text{alloc}(\varsigma) \)
\( \rho \in \text{Env} = \text{Var} \rightarrow \text{Addr} \)

\( \kappa \in \text{Kont} ::= [\ ] | \varphi : a \)

\( \sigma \in \text{Store} = \text{Addr} \rightarrow \wp(\text{Value} \times \text{Env} + \text{Kont}) \)

\( \langle x, \rho, \sigma, \kappa \rangle \mapsto \langle \text{addr}(\rho(x)), \sigma, \kappa \rangle \)

\( \langle (e_0 e_1), \rho, \sigma, \kappa \rangle \mapsto \langle e_0, \rho, \sigma', \text{ar}(e_1, \rho):a \rangle \quad \sigma' = \sigma \cup [a \mapsto \{\kappa\}] \)

\( \langle v, \rho, \sigma, \text{ar}(e, \rho):b \rangle \mapsto \langle e, \rho, \sigma, \text{fn}(v, \rho):b \rangle \)

\( \langle v, \rho, \sigma, \text{fn}(\lambda x.e, \rho') : b \rangle \mapsto \langle e, \rho'', \sigma', \kappa \rangle \quad \kappa \in \sigma(b) \)

where \( \rho'' = \rho'[x \mapsto a] \)

\( \sigma' = \sigma \cup [a \mapsto \text{force}(v)] \)

\( a = \text{alloc}(\varsigma) \)
(f x y)
$(f \ x \ y)$
1. Lazy non-determinism
2. Abstract compilation
3. Store deltas
\[ \rho \in \text{Env} = \text{Var} \rightarrow \text{Addr} \]
\[ \kappa \in \text{Kont} ::= [] | \phi : a \]
\[ \sigma \in \text{Store} = \text{Addr} \rightarrow \mathcal{P}(\text{Value} \times \text{Env} + \text{Kont}) \]
\[ \langle x, \rho, \sigma, \kappa \rangle \mapsto \langle v, \rho', \sigma, \kappa \rangle \]

\[ \text{if } (v, \rho') \in \sigma(\rho(x)) \]
\[ \langle (e_0 e_1), \rho, \sigma, \kappa \rangle \mapsto \langle e_0, \rho, \sigma', \text{ar}(e_1, \rho): a \rangle \sigma' = \sigma \cup [a \mapsto \{ \kappa \}] \]
\[ \langle v, \rho, \sigma, \text{ar}(e, \rho): b \rangle \mapsto \langle e, \rho, \sigma, \text{fn}(v, \rho): b \rangle \]
\[ \langle v, \rho, \sigma, \text{fn}(\lambda x. e, \rho'): b \rangle \mapsto \langle e, \rho'', \sigma', \kappa \rangle \kappa \in \sigma(b) \]

\[ \text{where } \rho'' = \rho'[x \mapsto a] \]
\[ \sigma' = \sigma \cup [a \mapsto \{(v, \rho)\}] \]

\[ a = \text{alloc}(\varsigma) \]
(match e ((var x) ((continue v σ κ) : v ∈ (get σ (get ρ x))))) ((}
Compile away dispatch overhead

...
〈e, ρ, σ, k〉
$[e](\rho, \sigma, \kappa)$
$\llbracket e \rrbracket(\rho, \sigma, \kappa)\langle e, \rho, \sigma, \kappa \rangle \mapsto \text{RHS}$

$\downarrow$

$\llbracket e \rrbracket = \lambda \rho, \sigma, \kappa. \text{RHS}$
(f x y)
\((f \times y)\)
1. Lazy non-determinism
2. Abstract compilation
3. Store deltas
Store widening / Global store

implimented as \( \text{step} : \text{State} \rightarrow \wp(\text{State}) \)

Lift and iterate to fixed point in \( \wp(\text{State}) \rightarrow \wp(\text{State}) \)

\[
\{ \langle e_0, \rho_0, \sigma_0, \kappa_0 \rangle, \ldots, \langle e_n, \rho_n, \sigma_n, \kappa_n \rangle \} \mapsto \{ \langle e'_0, \rho'_0, \sigma'_0, \kappa'_0 \rangle, \ldots, \langle e'_m, \rho'_m, \sigma'_m, \kappa'_m \rangle \} \]
Store widening / Global store

\[ \mapsto \text{implemented as } \text{step : State} \to \wp(\text{State}) \]

Lift and iterate to fixed point in \( \wp(\text{State}) \to \wp(\text{State}) \)

\[
\{ \langle e_0, \rho_0, \sigma_0, \kappa_0 \rangle, \ldots \} \mapsto \{ \langle e'_0, \rho'_0, \sigma'_0, \kappa'_0 \rangle, \ldots \}
\]

\[ \sigma_0 \cdots \sigma_n \mapsto \sigma'_0 \cdots \sigma'_m \]
Store widening / Global store

\[ \mapsto \text{implemented as } \text{step} : \text{State} \to \wp(\text{State}) \]

Lift and iterate to fixed point in \( \wp(\text{State}) \to \wp(\text{State}) \)

\[
\begin{align*}
\{ \langle e_0, \rho_0, \sigma_0, \kappa_0 \rangle, \ldots \} \quad &\mapsto \quad \{ \langle e'_0, \rho'_0, \sigma'_0, \kappa'_0 \rangle, \ldots \} \\
\{ \langle e_n, \rho_n, \sigma_n, \kappa_n \rangle \} \quad &\mapsto \quad \{ \langle e'_m, \rho'_m, \sigma'_m, \kappa'_m \rangle \} \\
\sigma_{\text{global}} &\quad \sigma'_{\text{global}}
\end{align*}
\]
Store widening / Global store

implied as \( \text{step} : \text{State} \rightarrow \wp(\text{State}) \)

Lift and iterate to fixed point in \( \wp(\text{State}) \rightarrow \wp(\text{State}) \)

\[
\{ \langle e_0, \rho_0, \sigma_0, \kappa_0 \rangle, \ldots, \langle e_n, \rho_n, \sigma_n, \kappa_n \rangle \} \quad \rightarrow \quad \{ \langle e'_0, \rho'_0, \sigma'_0, \kappa'_0 \rangle, \ldots, \langle e'_m, \rho'_m, \sigma'_m, \kappa'_m \rangle \} 
\]

\( \sigma_{\text{global}} \)

All small changes
Store widening / Global store

impl'd as \( \text{step} : \text{State} \rightarrow \wp(\text{State}) \)

Lift and iterate to fixed point in \( \wp(\text{State}) \rightarrow \wp(\text{State}) \)

\[
\left\{ \langle e_0, \rho_0, \sigma_0, \kappa_0 \rangle, \ldots, \langle e_n, \rho_n, \sigma_n, \kappa_n \rangle \right\} \rightarrow \left\{ \langle e'_0, \rho'_0, \sigma'_0, \kappa'_0 \rangle, \ldots, \langle e'_m, \rho'_m, \sigma'_m, \kappa'_m \rangle \right\}
\]

\( \sigma_{\text{global}} \)

\( \text{replay changes} \)
What does all this buy us?
1000000% improvement
Factor speed-up over naive vs. paper section

No change in evaluated precision
The evolution of AAM to OAAM
The evolution of AAM to OAAM
Before we part

Simple, systematic implementation

1000x speedup

"A simple matter of engineering?"

Build your tricks into the semantics

Thank you