Abstracting Abstract Control

J. Ian Johnson
ianj@ccs.neu.edu
Northeastern University
Boston, MA, USA

David Van Horn
dvanhorn@cs.umd.edu
University of Maryland
College Park, MD, USA
“Sounds abstract”
Abstract

We demonstrate a deconstruction approach to abstract interpretation that yields model and transparently sound static analysis when applied to well-established abstract machines. To demonstrate the technique and support our claim, we transform the CED machine of Feinleib and Friedson, a lazy variant of Kiviet’s machine, and the stack-abstracting CM machine of Clements and Feinleib into abstract interpretations of themselves. The resulting analyses bound temporal ordering of program events, predict reuse-flow and stack-injection 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 we call semi-abstracted continuations, leads to machines that abstract into static analyses simply by bundling their states. We demonstrate that the technique scales up sufficiently to allow static analyses of realistic language features, including tail calls, conditions, side effects, exceptions, first-class continuations, and even garbage collection.

Keywords: abstract machine, abstract interpretation

1. Introduction

Abstract machines such as the CED machine and Kiviet’s machine are first-order state transition systems that represent the core of a virtual machine implementation. Semantics-based program analyses, on the other hand, are concerned with safely approximating the behavior of such a machine as it runs a program. It seems natural then to want to systematically derive analyses from machines approximate the execution of realistic programs.

We introduce an interactive deconstruction approach to the generation of abstract interpretations of abstract machines for the purpose of modularly constructing deconstructible abstract interpretations of abstract machines by methods for transforming a given machine description into another that computes its abstract approximation.

Abstracting Control Machines

Oliveira Daze
Andreas Filinski
School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213, USA andreas@cs.cmu.edu

Introduction

Stacker and Wadsworth’s continuations were a breakthrough in understanding imperative constructs of programming languages. They gave a clear and unambiguous semantics to a wide class of control operations such as escapes and continuations. In recent years, however, there has been a growing interest in a class of control operators [Krivine, et al. 97] [Pierce 96] which do not seem to fit into this framework. The point of these new operators is to abstract, with regular procedures that do not escape when they are applied. This approach encourages writing not only procedures as the computational counterpart of functions but extending this view to continuations as well. However, the published approaches differ greatly from one another. Therefore, this work aims to represent continuations as functions of a considerable expressiveness and activations that pass the subsumption test of the original functional semantics. Does this mean that control operators substantially more powerful than those are indeed the limit of a traditional continuation semantics?

In the following, we present a denotational “standard semantics” [Milner & Stoller 78], where continuations are represented with functions and control is abstracted with procedures and others programs how natural, purely functional counterparts. In doing so, we replace the fundamentally dynamic control behavior specified by detailed specifications of combinators with a properly static approach, akin to the differences between flow and type.

The new idea is that a term is evaluated in a collection of extended calculi, each represented by a continuation. The denotations of a term is expressed in a functional continuation-passing style (CPS). Essentially, our proposal is to model continuations and control operators as combinators. This can be achieved, for example, by replacing a call-by-value evaluation with a call-by-need evaluation of the CPS continuation of the term.

The new proposal is a small step towards the realization of a theoretical framework for the study of control operators. It opens up the possibility of using control operators as a tool for programming and reasoning about programs in a way that is consistent with the classical semantics of functional programming.
"Sounds abstract"

Abstract

We demonstrate that the technique of reconfiguring a machine with non-allocate continuations allows a direct structural abstraction of by bounding the machine's store. Thus, we are able to convert semantic techniques used to model language features into static analyzers for reasoning about the behavior of those very same features. By abstracting well-known machines, our technique derives static analyzers that can reason about first-order evaluation, higher-order functions, tail calls, side effects, stack structure, exceptions and free-class continuations.

Categories and Subject Descriptions F.3.2 [Logics and Meanings of Programs]; Semantics of Programming Languages—Program analysis, Operational semantics; F.1.4 [Mathematical Logic and Formal Languages]: Mathematical Logic—Lambda calculus and related systems

General Terms Languages, Theory

Keywords abstract machines, abstract interpretation

1. Introduction

Abstract machines such as the CEK machine and Keviczky’s machine are first-order state transition systems that represent the core of a virtual machine implementation. Semantics-based program analysis, on the other hand, is concerned with safely approximating intensional properties of such a machine as it runs a program. It seems natural then to want to systematically derive analyzers from machines to approximate the core of real-life execution systems.

We present a new approach to abstracting interpretations of abstract machines by methods for transforming a given machine description into another that computes its trace approximation.

*Supported by the National Science Foundation under grant (8717090) to the Computing Research Association for the Officers' Project EOP-1P, September 7-27, 1987. Baltimore, Maryland, USA.

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 the 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.\n
Copyright \( \odot 1988 \) ACM \( \text{840-0365/88/1100...} \) \$1.50

Abstract Models of Management Memory

Greg Morrisett

Matthias Felleisen

Robert Harper

A Tail-Recursive Machine with Stack Inspection

JOHN CLEMENTS and MATTHIAS FELLEISEN

Pushdown Flow Analysis of First-Class Control

Dimitrios Vardoulakis

Olin Shivers

Northwestern University
dsvard@cac.nwu.edu
shivers@cac.nwu.edu

Abstract

Pushdown models are better than control-flow graphs for higher-order flow analysis. They faithfully model the call stack structure of a program, which results in fewer spurious edges and increased precision. However, pushdown models require that calls and returns in the analyzed program are proper. As a result, they cannot be used to analyze language constructs that break call stack nesting such as generators, coroutines, co-routines, etc.

In this paper, we extend the CPA flow analyzer to analyze the fr pushdown flow graphs for languages with first-class control. We describe the CPA flow analyzer, and extend it to analyze pushdown models. We then present a new pushdown flow analysis algorithm that handles the classes of constructs described above.

Categories and Subject Descriptions F.3.2 [Semantics of Programming Languages]: Program Analysis

General Terms Languages

Keywords pushdown flow analysis, first-class continuations, return stack analysis
Abstracting Abstract Machines

Interpreter

Abstract interpreter
Abstracting Abstract Machines

Interpreter

→ Abstract interpreter

Everything is an abstract interpretation!
Everything is an abstract interpretation!
Flow analysis
Symbolic evaluator
Termination/productivity analysis
White-box fuzzer
Abstracting Abstract Machines

Interpreter

Allocator

Abstract interpreter
Abstracting Abstract Machines

Interpreter

Allocator

Abstract interpreter
Abstracting Abstract Machines

Interpreter

Allocator

Con: loses important control structure
Who cares about continuations?
Who cares about continuations?
Who cares about continuations?

RESTful web applications
Event-driven programming
Cloud computing
Actors
Operating systems
(Game engines?)
Who cares about continuations?

REST
Event-driven programming
Cloud computing
Actors
Operating systems

Typesafe

Hekate — a highly-concurrent BitTorrent seeder.
Who cares about continuations?

RESTful web applications

Event-driven programming

Cloud computing

Actors

Operating systems

(Game engines?)

I don't understand!

Hekate — a highly-concurrent BitTorrent seeder.
</motivation>
s \mapsto s'}
Heap-allocate recursion

\[ s \mapsto s' \quad \hat{s} \mapsto \hat{s}' \]
Heap-allocate recursion

\[ \langle \text{code, heap, cont} \rangle \]
Heap-allocate recursion

\langle \text{code, heap, cont} \rangle
Heap-allocate recursion

\[
\langle \text{code, heap, cont} \rangle
\]

\[
s \mapsto s' \quad \hat{s} \mapsto \hat{s}'
\]

cont : List[Activation-Frame]
Heap-allocate recursion

\[ s \mapsto s' \quad \hat{s} \mapsto \hat{s}' \]

\[ \text{cont} : \text{List[Activation-Frame]} \]

\[ \text{cons} : X \rightarrow \text{List}[X] \rightarrow \text{List}[X] \]
Heap-allocate recursion

\[ s \mapsto s', \hat{s} \mapsto \hat{s}' \]

cont : List[Activation-Frame]

cons : X \rightarrow Addr \rightarrow List[X]
Heap-allocate recursion

\( \langle \text{code, heap, cont} \rangle \)

\[ s \mapsto s' \quad \# \quad \hat{s} \mapsto \hat{s}' \]

\( \text{cont} : \text{List}[(\text{Activation-Frame})] \)

\( \text{cons} : X \rightarrow \text{Addr} \rightarrow \text{List}[X] \)

\( \text{heap} : \text{Map}[\text{Addr, Value}] \)
Heap-allocate recursion

\[ s \mapsto s', \hat{s} \mapsto \hat{s}' \]

\[ \text{cont} : \text{List[Activation-Frame]} \]
\[ \text{cons} : X \rightarrow \text{Addr} \rightarrow \text{List[X]} \]
\[ \text{heap} : \text{Map[Addr, Set[Value]]} \]
\[ h[a \mapsto v] \Rightarrow h[a \mapsto h(a) \cup \{v\}] \]
Say we have some function $f : \text{json} \rightarrow \text{html}$
Say we have some function $f: \text{json} \rightarrow \text{html}$

We wrap it to validate its input and output

$$(\lambda \ (j) \ (\text{if}\ (\text{good-json?} \ j) \ (\text{let}\ ([\ r\ (f\ j)])\ (\text{if}\ (\text{good-html?} \ r)\ r\ (\text{blame}\ 'f)))\ (\text{blame}\ 'user))))$$
Say we have some function \( f : \text{json} \rightarrow \text{html} \)

We wrap it to validate its input and output

\[
(\lambda (j) \\
  \quad \text{if} \ (\text{good-json?} \ j) \\
  \quad \quad (\text{let} \ ([r \ (f \ j)]) \\
  \quad \quad \quad \quad (\text{if} \ (\text{good-html?} \ r) \\
  \quad \quad \quad \quad \quad \quad r \\
  \quad \quad \quad \quad \quad \quad (\text{blame 'f}))) \\
  \quad \quad \quad \quad (\text{blame 'user}))) \\
\]

\[
(\text{document.write} \ (\text{p}, (\text{read-request} \ f) \ , (\text{read-request} \ f))) \\
\]

\text{read-request} blocks until json is read, then calls f
Say we have some function \( f : \text{json} \rightarrow \text{html} \)

We wrap it to validate its input and output

\[
(\lambda \ j \to \\
  (\text{if} \ (\text{good-json?} \ j) \\
   (\text{let} \ (\text{list} \ [(r \ (f \ j)])) \\
    (\text{if} \ (\text{good-html?} \ r) \\
     r \\
     (\text{blame} \ 'f))) \\
    (\text{blame} \ '\text{user})))
\]

\text{read-request} blocks until json is read, then calls f
Say we have some function \( f : \text{json} \rightarrow \text{html} \)

We wrap it to validate its input and output

\[
(\lambda (j) \quad 
  (\text{if} \ (\text{good-json?} \ j) \quad 
    (\text{let} \ ([r \ (f \ j)]) \quad 
      (\text{if} \ (\text{good-html?} \ r) \quad 
        r \quad 
        (\text{blame} \ 'f))) \quad 
        (\text{blame} \ 'user))) \quad 
      (\text{document.write} \ `(p , (\text{read-request} \ f) , (\text{read-request} \ f)))
\]

\text{read-request} blocks until json is read, then calls \( f \)
Say we have some function \( f : \text{json} \rightarrow \text{html} \)

We wrap it to validate its input and output

\[
\lambda (j).
\begin{cases}
  \text{if } (\text{good-json? } j) \\
  \quad \text{let } ([r \ (f \ j)]) \\
  \quad \quad \text{if } (\text{good-html? } r) \\
  \quad \quad \quad r \\
  \quad \quad \quad (\text{blame 'f})) \\
  \quad \quad \quad (\text{blame 'user}))
\end{cases}
\]

\[
\text{document.write ´(p ,(read-request f) ,,(read-request f))}
\]

\texttt{read-request} blocks until json is read, then calls \( f \)
Say we have some function \( f : \text{json} \rightarrow \text{html} \)

We wrap it to validate its input and output

\[
(\lambda (j) \text{ if } (\text{good-json? } j) \text{ let } ([r (f j)]) \text{ if } (\text{good-html? } r) r \text{ blame } 'f))) \text{ blame } '\text{user})))
\]

\[
(\text{document.write } `(p , (\text{read-request } f) , (\text{read-request } f))
\]

\text{read-request} blocks until json is read, then calls \( f \)
Say we have some function \( f : \text{json} \rightarrow \text{html} \)

We wrap it to validate its input and output

\[
\lambda (j) \quad \begin{cases} 
(\text{if} \ (\text{good-json?} \ j) \\
(\text{let} \ ([r \ (f \ j)]) \\
(\text{if} \ (\text{good-html?} \ r)\\
\quad r \\
(\text{blame 'f}))) \\
(\text{blame 'user}))) \\
(\text{document.write `(~p ,(read-request f) , (read-request f)}) \\
\end{cases}
\]

\text{read-request} blocks until json is read, then calls \( f \)
Say we have some function $f : \text{json} \rightarrow \text{html}$

We wrap it to validate its input and output

$$(\lambda \ (j) \ (\text{if} \ (\text{good-json?} \ j) \ (\text{let} \ ([r \ (f \ j)]) \ (\text{if} \ (\text{good-html?} \ r) \ r \ (\text{blame} \ 'f)))) \ (\text{blame} \ '\text{user})))) \ (\text{document.write} \ `(p , (\text{read-request} \ f) , (\text{read-request} \ f))))$$

$\text{read-request}$ blocks until json is read, then calls $f$
Say we have some function \( f : \text{json} \rightarrow \text{html} \)

We wrap it to validate its input and output:

\[
(\lambda (j) \\
\quad \text{(if (good-json? j)} \\
\quad \quad \text{(let ([r (f j)]))} \\
\quad \quad \quad \text{(if (good-html? r) r)} \\
\quad \quad \quad \quad \text{(blame 'f)))} \\
\quad \quad \text{(blame 'user))})
\]

\( \text{document.write `(p ,(read-request f)} \)
\quad \text{,(read-request f)))} \)

\text{read-request} blocks until json is read, then calls \( f \)
Insight:

delimit computations &
catalog contexts by relevant state
The stack doesn’t matter*

*yet
\((\lambda \ (j)\  
  \text{if} \ (\text{good-json?} \ j)\  
  (\text{let} \ ([r \ (f \ j)])\  
    (\text{if} \ (\text{good-html?} \ r)\  
     r\  
     (\text{blame} \ 'f)))\  
  (\text{blame} \ 'user)))\  
  (\text{document.write} \ `\((p \ ,(\text{read-request} \ f)\  
    ,(\text{read-request} \ f))\)`)\)
(λ (j) (if (good-json? j)
  (let ([r (f j)])
    (if (good-html? r)
      r
      (blame 'f)))))

(document.write `(p , (read-request f), (read-request f)))

Contexts = [• ➞ {cont}]
\[
\lambda (j) \quad \text{if} \quad \text{good-json?} \quad j \quad \text{let} \quad ([r (f \ j)]) \quad \text{if} \quad \text{good-html?} \quad r \quad r \quad \text{blame 'f)}))) \quad \text{blame 'user})) \quad \text{document.write} \quad `(p , (read-request \ f), (read-request \ f))
\]

Contexts = [\bullet \leftrightarrow \{\text{cont}\}, \circ \leftrightarrow \{\text{cont}\}]

What’s really going on here?

AAM told us cons : X -> Addr -> List[X]
What’s really going on here?

AAM told us cons : X -> Addr -> List[X]

Are just fancy addresses?
What’s really going on here?

AAM told us \( \text{cons} : X \rightarrow \text{Addr} \rightarrow \text{List}[X] \)

Are just fancy addresses?

States are \( \langle \text{code heap stack} \rangle \) and the stack is irrelevant

is \( \langle \text{code heap} \rangle \)
What’s really going on here?

AAM told us \( \text{cons : } X \rightarrow \text{Addr} \rightarrow \text{List}[X] \)

Are \( \bullet \) just fancy addresses?

States are \( \langle \text{code heap stack} \rangle \) and the stack is irrelevant

\( \bullet \) is \( \langle \text{code heap} \rangle \)

\[ h[\langle c,h' \rangle \mapsto \{\text{cont}\}] \]
What’s really going on here?

AAM told us

\[ \text{cons} : X \rightarrow \text{Addr} \rightarrow \text{List}[X] \]

Are \(\bullet\) just fancy addresses?

States are \(\langle \text{code heap stack} \rangle\) and the stack is irrelevant

\[ \bullet \text{ is } \langle \text{code heap} \rangle \]

\[ h[\langle c, h' \rangle] \mapsto \{\text{cont}\} \]
What's really going on here?

AAM told us \( \text{cons} : X \rightarrow \text{Addr} \rightarrow \text{List}[X] \)

Are \( \bullet \) just fancy addresses?

States are \( \langle \text{code heap stack} \rangle \) and the stack is irrelevant

\( \bullet \) is \( \langle \text{code heap} \rangle \)

\[
\begin{align*}
\text{h}[\langle c, h' \rangle] & \mapsto \{\text{cont}\}
\end{align*}
\]

\( \bullet \) are stored in a stratified heap: Contexts
What if “the stack” isn’t a stack?
What if “the stack” isn’t a stack?

\[ E[F[(\text{shift } k \ e)]] \mapsto E[e\{k := (\lambda (x) F[x])\}] \]
\[ E[F[(\text{shift } k \ e)]] \mapsto E[e\{k := (\lambda (x) F[x])\}] \]

\[ (+ 10 \ (\text{reset} \ (+ 2 \ (\text{shift } k \ (+ 40 \ (k \ (k \ 3)))))))))) \]
\( E[F[(\text{shift } k \ e)]] \mapsto E[e\{k := (\lambda \ (x) \ F[x])\}] \)

\[
(+ 10 \ (\text{reset} \ (+ 2 \ (\text{shift } k \ (+ 40 \ (k \ (k \ 3)))))))
\]
\[ \mathbb{E}[F[(\text{shift } k \ e)]] \mapsto \mathbb{E}[e\{k := (\lambda (x) F[x])\}] \]

now a function

\[ (+ 10 (\text{reset} (+ 2 (\text{shift } k (+ 40 (k (k 3)))))))) \]
\( E[F[(\text{shift } k \ e)]] \mapsto E[e\{k := (\lambda \ (x) \ F[x])\}] \)

now a function

run from here

(\(+ \ 10 \ (\text{reset} \ (+ \ 2 \ (\text{shift } k \ (+ \ 40 \ (k \ (k \ 3))))))))\)
\[ E[F[(\text{shift } k \ e)]] \mapsto E[e\{k := (\lambda (x) \ F[x])\}] \]

now a function

run from here

(+ 10 (reset (+ 2 (shift k (+ 40 (k (k 3)))))))

(+ 2 [])
\[ E[F[(\text{shift } k \ e)]] \mapsto E[e\{k := (\lambda (x) F[x])\}] \]

now a function

run from here

(+ 10 (reset (+ 2 (shift k (+ 40 (k (k 3)))))

(+ 2 [])

(+ 10 (+ 40 (k (k 3))))

(+ 10 (reset (+ 2 (shift k (+ 40 (k (k 3)))))

(+ 2 []))

(+ 10 (+ 40 (k (k 3)))))
\[ E[F[(\text{shift } k \ e)]] \leftrightarrow E[e\{k := (\lambda \ (x) \ F[x])\}] \]

now a function

\[ (+ \ 10 \ (\text{reset} \ (+ \ 2 \ (\text{shift} \ k \ (+ \ 40 \ (k \ (k \ 3))))))) \]

k = (\lambda \ (x) \ (+ \ 2 \ x))

(+ 10 (+ 40 (k (k 3))))

run from here
\[ E[F[(\text{shift } k \ e)]] \mapsto E[e\{k := (\lambda (x) F[x])\}] \]

\[
(+ 10 (\text{reset} (+ 2 (\text{shift } k (+ 40 (k (k 3)))))))
\]

\[ k = (\lambda (x) (+ 2 x)) \]

\[
(+ 10 (+ 40 ((\lambda (x) (+ 2 x)) ((\lambda (x) (+ 2 x)) 3)))
\]
\[ E[F[(\text{shift } k \ e)]] \mapsto E[e\{k := (\lambda (x) F[x])\}] \]

```
(+ 10 (reset (+ 2 (shift k (+ 40 (k (k 3))))))))
```

\[ k = (\lambda (x) (+ 2 \ x )) \]

\[ (+ 10 (+ 40 (+ 2 (+ 2 3)))) \]

now a function

run from here
\[
E[F[(\text{shift} \ k \ e)]] \leftrightarrow E[e\{k := (\lambda \ (x) \ F[x])\}]
\]

now a function

\[
(+ \ 10 \ (\text{reset} \ (+ \ 2 \ (\text{shift} \ k \ (+ \ 40 \ (k \ (k \ 3)))))))
\]

\[
k = (\lambda \ (x) \ (+ \ 2 \ x))
\]

\[
(+ \ 10 \ (+ \ 40 \ (+ \ 2 \ (+ \ 2 \ 3))))
\]

57
\((\lambda \ (j)\n  \quad (if \ (good-json? \ j)\n      \quad (let \ ([r \ (f \ j)])\n          \quad (if \ (good-html? \ r)\n              \quad r\n              \quad (blame \ 'f])))\n      \quad (blame \ 'user)))\n\)

(document.write `\(p \ ,\(\text{read-request} \ f)\n      \quad ,\(\text{read-request} \ f))\n\)

\textbf{read-request} uses non-blocking I/O
\( \lambda (i) \)

\[
\begin{align*}
&(\text{let} \quad [r (f j)])
\quad \text{if} \quad \text{good-html?} (r)
\quad \text{r} (\text{blame} \ 'f')) \\
&\quad \text{blame} \ 'user'))
\end{align*}
\]

\( \text{document.write} \ `(p , (\text{read-request} \ f) , (\text{read-request} \ f)) \)

\texttt{read-request} uses non-blocking I/O
\( \lambda \) 

(\lambda (i) 

\begin{array}{l}
  \text{(define (read-request f)} \\
  \quad \text{(shift k (evloop-until-evt)} \\
  \quad \quad \text{(read-request-evt f)} \\
  \quad \quad \text{k)})) \\
  \text{(blame 'f))} \\
  \text{(blame 'user))} \\
\end{array}

\text{(document.writeln (read-request f) (read-request f))}

\text{read-request uses non-blocking I/O}
\[(\lambda j.\ (\text{if good-json?} j\ (\text{let } (r (f j))\ (\text{if good-html?} r\ r (\text{blame 'f}))\ (\text{blame 'user}))))\]

\text{read-request} uses non-blocking I/O

\texttt{(define (read-request f)}
\texttt{  (shift k (evloop-until-evt}
\texttt{    (read-request-evt f)
\texttt{      k)))}
\texttt{  (blame 'f))}
\texttt{  (blame 'user))}

\texttt{(document.}
\texttt{  h[ka \mapsto \{(comp \langle c,h'\rangle\}]}
\texttt{read-request))}
(λ (i)

(define (read-request f)
  (shift k (evloop-until-evt
      (read-request-evt f) k))
  (h[k ↦ {((comp <c,h'>))}])))

Can we stratify like with Contexts?

read-request uses non-blocking I/O
Of course not!

\( \langle \text{shift } k \ e \rangle, \text{heap}, \bullet \rangle \) produces \( \text{heap}(ka) \ \exists (\text{comp } \langle c,a \rangle) \)
Of course not!

\[ \langle \text{shift } k e \rangle, \text{heap, } \bullet \rangle \text{ produces } \text{heap}(ka) \ni (\text{comp } \langle c, a \rangle) \ni \chi(a) \ni h' \]
Of course not!

\[ \langle \text{shift } k \text{ e}, \text{ heap, } \bullet \rangle \text{ produces } \text{heap}(ka) \ni (\text{comp } \langle c, a \rangle) \]

\[ \chi(a) \ni h' \]

Well, now \( \chi \) is relevant!
Of course not!

\[ \langle \text{shift } k \ e \rangle, \text{heap, } \bullet \rangle \text{ produces } \text{heap}(ka) \ \exists \ (\text{comp } \langle c, a \rangle) \]

\[ \chi(a) \ \exists \ h' \]

Well, now \( \chi \) is relevant! Since \( \chi \) closes the heap
Of course not!

\langle (\text{shift } k \text{ e}), \text{heap}, \bullet \rangle \text{ produces } \text{heap}(ka) \ni (\text{comp } \langle c, a \rangle)

\chi(a) \ni h'

Well, now \chi is relevant! Since \chi closes the heap

\bullet \equiv \langle c', h', \chi' \rangle
Of course not!

\[ \langle \text{shift } k \ e \rangle, \text{heap}, \bullet \rangle \text{ produces heap}(ka) \ni (\text{comp } \langle c, a \rangle) \]

\[ \chi(a) \ni h' \]

Well, now \( \chi \) is relevant! Since \( \chi \) closes the heap

\[ \bullet \equiv \langle c', h', \chi' \rangle \]

\[ \chi(a) \ni \langle h', \chi' \rangle \]
Of course not!

\[ \langle \text{shift } k \ e \rangle, \text{heap}, \bullet \rangle \text{ produces } \text{heap}(ka) \ni \text{(comp } \langle c, a \rangle) \]

\[ \chi(a) \ni h' \]

Well, now \( \chi \) is relevant! Since \( \chi \) closes the heap

\[ \bullet \equiv \langle c', h', \chi' \rangle \]

\[ \chi(a) \ni \langle h', \chi' \rangle \]
Of course not!

\[ \langle \text{shift k e}, \text{heap}, \bullet \rangle \text{ produces } \text{heap}(ka) \ni (\text{comp } \langle c, a \rangle) \]

\[ \chi(a) \ni h' \]

Well, now \( \chi \) is relevant! Since \( \chi \) closes the heap \( \equiv \langle c', h', \chi' \rangle \)

\[ \chi(a) \ni \langle h', \chi' \rangle \]

\( \chi \) and heap are mutually recursive! Can't stratify!
Squash it

Instead of $\chi \sqcup [a \mapsto \langle h', \chi' \rangle]$

we do $\chi \sqcup \chi' \sqcup [a \mapsto \{h'\}]$

$\llbracket \langle c',a \rangle \rrbracket = \{\text{cont} \in \text{Contexts}(\langle c',h',\chi' \rangle) : h' \in \chi(a), \chi' \sqsubseteq \chi\}$
\[
\lambda j \ (\text{if} \ \text{good-json?} \ j \ \text{let} \ ([r (\text{f} j)]) \ (\text{if} \ \text{good-html?} \ r \ (\text{blame 'f'))) \ (\text{blame 'user})))
\]

\(h = []\)

\(\chi = []\)
(define (read-request f)
  (λ (j)
      (if (good-json? j)
          (let ([r (f j)])
              (if (good-html? r)
                  r
                  (blame 'f)))
          (blame 'user))))

(h = [ka ↦ {(comp ⟨•,a⟩)}])

(χ = • ∪ [a ↦ {•}])
(define (read-request f)
  (λ k
    (shift k (evloop-until-evt
               (read-request-evt f)
               k)))))

(if (good-html? r)
    r
    (blame 'f)))

h = [ka ↦ {(comp ⟨1,a⟩), (comp ⟨1,a⟩)}]

χ = ⊔ ⊔ [a ↦ {♠, ♦}]

(document (read-request f)
           (read-request f))
(define (read-request f)
  (λ k
    (shift k (evloop-until-evt
      (read-request-evt f) k))))

(if (good-html? r)
  r
  (blame 'f)))

h = [ka ↦ {(comp 〈1,a〉)},
     ka ↦ {(comp 〈2,a〉)}]

χ = ϕ ⊔ ϕ ⊔ [a ↦ {b}] ⊔ [a ↦ {d}]
Where do we stand?

abstract languages and respect control
Where do we stand?

abstract languages and respect control

Want shift/reset in modular semantics
Where do we stand?

abstract languages and respect control

Want shift/reset in modular semantics

(what if (comp ●) is...?)
Where do we stand?

abstract languages and respect control

Want shift/reset in modular semantics

(what if (comp ○) is)

Not all the heap is relevant
Where do we stand?

abstract languages and respect control

Want shift/reset in modular semantics

(what if (comp •) is)

Not all the heap is relevant [Stefan Staiger-Stöhr diss]
Takeaway

Delimit computations by relevant state
Takeaway

Delimit computations by relevant state

Squash abstracted relevance objects
Takeaway

Delimit computations by relevant state

Squash abstracted relevance objects

Break cycles in state space with addresses
Delimit computations by relevant state

Squash abstracted relevance objects

Break cycles in state space with addresses

Thank you