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”

“Sounds abstract”

Abstracting Abstract Machines

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

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

Abstract

We describe a derivational approach to abstract interpretation that yields novel and transparently sound static analyses when applied to well-established abstract machines. 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 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.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis, Operational semantics; F.4.1 [*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 Krivine's machine are first-order 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 intensional properties of such a machine as it runs a program. It seems natural then to want to systematically derive analyses from machines to approximate the core of realistic run-time systems.

Our goal is to develop a technique that enables direct abstract interpretations of abstract machines by methods for transforming a given machine description into another that computes its finite approximation.

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

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

We demonstrate that the technique of refactoring a machine with **store-allocated continuations** allows a direct structural abstraction[1] 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 those very same features. By abstracting well-known machines, our technique delivers static analyzers that can reason about by-need evaluation, higher-order functions, tail calls, side effects, stack structure, exceptions and first-class continuations.

The basic idea behind store-allocated continuations is not new. SML/NJ has allocated continuations in the heap for well over a decade [28]. 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.

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 [13],
- a call-by-need λ-calculus based on a tail-recursive, lazy variant of Krivine's machine derived by Ager, Danvy and Midtgaard [1], and
- a call-by-value λ-calculus with stack inspection based on the CM machine of Clements and Felleisen [3];

and use abstract garbage collection to improve precision [25].

Overview

In Section 2, 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.

[1] A structural abstraction distributes component-, point-, and member-wise.

Abstracting Control *

Olivier Danvy †

Andrzej Filinski

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
andrzej+@cs.cmu.edu

Abstract

The last few years have seen a renewed interest in continuations for expressing advanced control structures in programming languages, and new models such as Abstract Continuations have been proposed to capture these dimensions. This article investigates an alternative formulation, exploiting the latent expressive power of the standard continuation-passing style (CPS) instead of introducing yet other new concepts. We build on a single foundation: abstracting control as a *hierarchy* of continuations, each one modeling a specific language feature as acting on nested *evaluation contexts*.

We show how *iterating* the continuation-passing conversion allows us to specify a wide range of control behavior. For example, two conversions yield an abstraction of Prolog-style backtracking. A number of other constructs can likewise be expressed in this framework; each is defined independently of the others, but all are arranged in a hierarchy making any interactions between them explicit.

This approach preserves all the traditional results about CPS, *e.g.*, its evaluation order independence. Accordingly, our semantics is directly implementable in a call-by-value language such as Scheme or ML. Furthermore, because the control operators denote simple, typable lambda-terms in CPS, they themselves can be statically typed. Contrary to intuition, the iterated CPS transformation does not yield huge results: except where explicitly needed, all continuations beyond the first one disappear due to the extensionality principle (η-reduction).

Besides presenting a new motivation for control operators, this paper also describes an improved conversion into applicative-order CPS. The conversion operates in one pass by performing all administrative reductions at translation time; interestingly, it can be expressed very concisely using the new control operators. The paper also presents some examples of nondeterministic programming in direct style.

*The core of this work was developed at DIKU, the Computer Science department at the University of Copenhagen, Denmark (danvy@diku dk, andrzej@diku dk)

†This work has benefited from visits to the Computer Science departments of Stanford University (thanks to Carolyn L Talcott), Indiana University (thanks to Daniel P Friedman), and Kansas State University (thanks to David A Schmidt) during the academic year 1989-1990

Introduction

Strachey 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 coroutines. In recent years, however, there has been a growing interest in a class of control operators [Felleisen *et al.* 87] [Felleisen 88] which do not seem to fit into this framework. The point of these new operators is to abstract control with regular procedures that do not escape when they are applied.

This approach encourages seeing not only procedures as the computational counterpart of functions but extending this view to continuations as well. However, the published semantic descriptions, [Felleisen *et al.* 88] do not actually represent continuations as functions but as concatenable sequences of activation frames, losing the inherent simplicity of the original functional formalism. Does this mean that control operators substantially more powerful than jumps are indeed beyond the limit of a traditional continuation semantics?

In the following, we present a denotational “standard semantics” [Milne & Strachey 76], where continuations are represented with functions and control is abstracted with procedures, and where programs have natural, purely functional counterparts. In doing so, we replace the fundamentally dynamic control scoping specified by prior definitions of composable continuations with a properly static approach, akin to the difference between Lisp and Scheme.

The new idea is that a term is evaluated in a collection of embedded contexts, each represented by a continuation. The denotation of a term is expressed in *extended continuation-passing style (ECPS)*. Essentially, this generalizes ordinary continuation-passing style to a hierarchy of continuations, one for each context. Very importantly, however, it inherits the characteristic, syntactically restricted form of a λ-calculus without nested function applications. As such, it still yields semantic specifications where the evaluation order of the defined language is independent of the evaluation order of the defining one [Reynolds 72].

Of course, extended continuation-passing style is in general more verbose than plain continuation-passing style. This suggests introducing new control operators to retain the ability of expressing programs in direct style, mirroring the rationale for including `call-with-current-continuation` in Scheme [Rees & Clinger 86] [Miller 87, appendix A]. We will show how such control operators can in fact be systematically added to an applicative order λ-calculus.

“Sounds abstract”


Abstracting Abstract Machines

David Van Horn [*]
Northeastern University
dvanhorn@ccs.neu.edu

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

Abstract

We describe a derivational approach to abstract interpretation that yields novel and transparently sound static analyses when applied to well-established abstract machines. 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 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.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis, Operational semantics; F.4.1 [*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 Krivine's machine are first-order 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 intensional properties of such a machine as it runs a program. It seems natural then to want to systematically derive analyses from machines to approximate the core of realistic run-time systems.

Our goal is to develop a technique that enables direct abstract interpretations of abstract machines by methods for transforming a given machine description into another that computes its finite approximation.

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

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-60558-794-3/10/09…$10.00

We demonstrate that the technique of refactoring a machine with **store-allocated continuations** allows a direct structural abstraction[1] 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 those very same features. By abstracting well-known machines, our technique delivers static analyzers that can reason about by-need evaluation, higher-order functions, tail calls, side effects, stack structure, exceptions and first-class continuations.

The basic idea behind store-allocated continuations is not new. SML/NJ has allocated continuations in the heap for well over a decade [28]. 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.

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 [13],
- a call-by-need λ-calculus based on a tail-recursive, lazy variant of Krivine's machine derived by Ager, Danvy and Midtgaard [1], and
- a call-by-value λ-calculus with stack inspection based on the CM machine of Clements and Felleisen [3];

and use abstract garbage collection to improve precision [25].

Overview

In Section 2, 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.

[1] A structural abstraction distributes component-, point-, and member-wise.


Abstracting Control *

Olivier Danvy †   Andrzej Filinski

Abstract Models of Memory Management*

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
Northeastern University
dimvar@ccs.neu.edu   shivers@ccs.neu.edu

Abstract

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

In this paper, we extend the CFA2 flow analysis to create the first pushdown flow analysis for languages with first-class control. We modify the abstract semantics of CFA2 to allow continuations to escape to, and be restored from, the heap. We then present a summarization algorithm that handles escaping continuations via a new kind of summary edges. We prove that the algorithm is sound with respect to the abstract semantics.

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

General Terms Languages

Keywords pushdown flow analysis, first-class continuations, restricted continuation-passing style, summarization

allow complex control flow, such as jumping back to functions that have already returned. Continuations come in two flavors. Undelimited continuations (`call/cc` in Scheme [19] and SML/NJ [5]) capture the entire stack. Delimited continuations [7, 9] [15, Scala 2.8] capture part of the stack. Continuations can express generators and coroutines, and also multi-threading [17, 24] and Prolog-style backtracking. All these operators provide a rich variety of control behaviors. Unfortunately, we cannot currently use pushdown models to analyze programs that use them.

We rectify this situation by extending the CFA2 flow analysis [21] to languages with first-class control. We make the following contributions.

- CFA2 is based on abstract interpretation of programs in continuation-passing style (*abbrev.* CPS). We present a CFA2-style abstract semantics for Restricted CPS, a variant of CPS that allows continuations to escape but also permits effective reasoning about the stack [23]. When we detect a continuation that may escape, we copy the stack into the heap (sec. 4.3). We prove that the abstract semantics is a safe approximation of the actual runtime behavior of the program (sec. 4.4).

Abstracting Abstract Machines

Interpreter

Abstract interpreter

Abstracting Abstract Machines

Interpreter

Abstract interpreter

Abstracting Abstract Machines

Interpreter

Abstract interpreter

Everything is an abstract interpretation!

Flow analysis

Symbolic evaluator

Termination/productivity analysis

White-box fuzzer

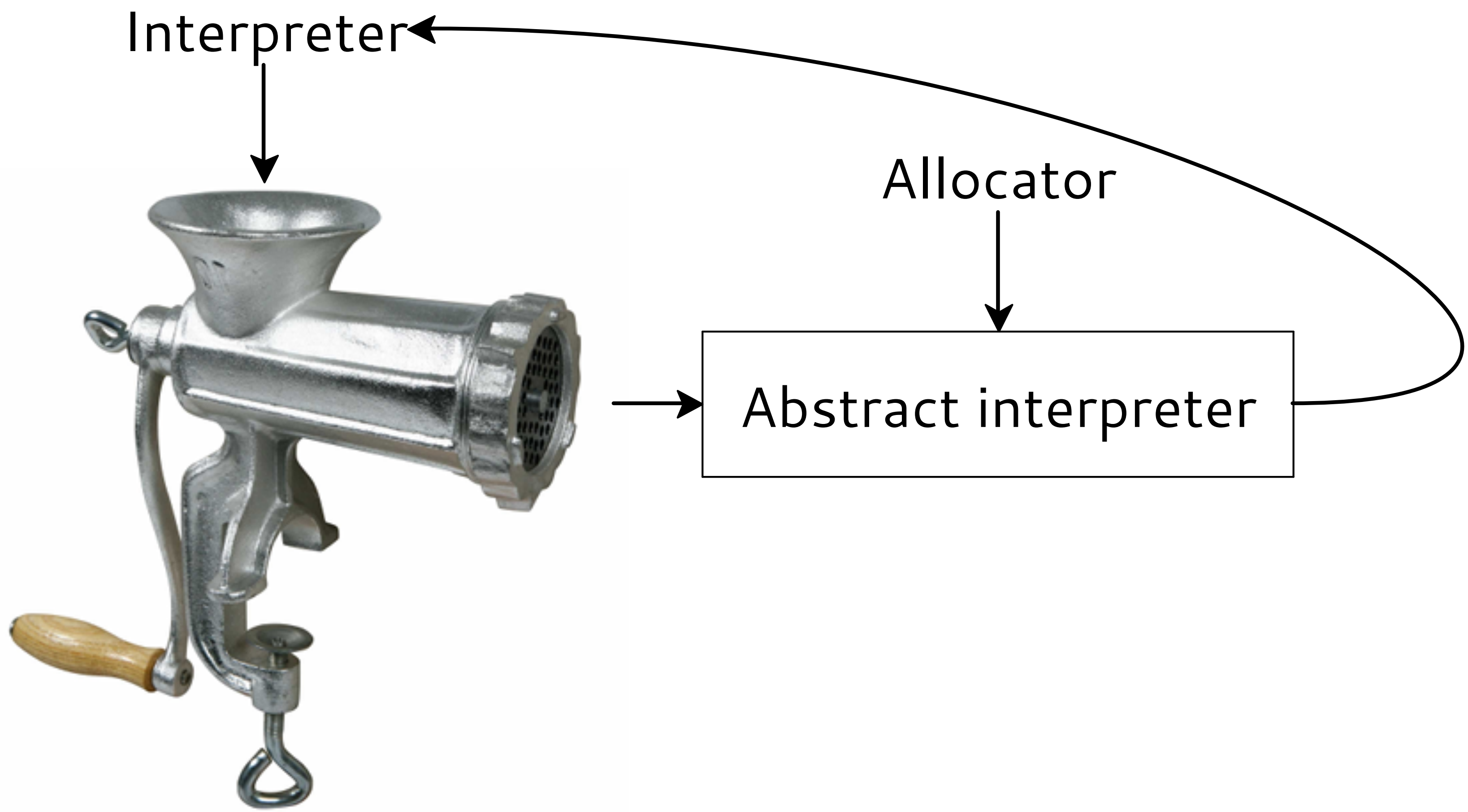
Abstracting Abstract Machines

Interpreter

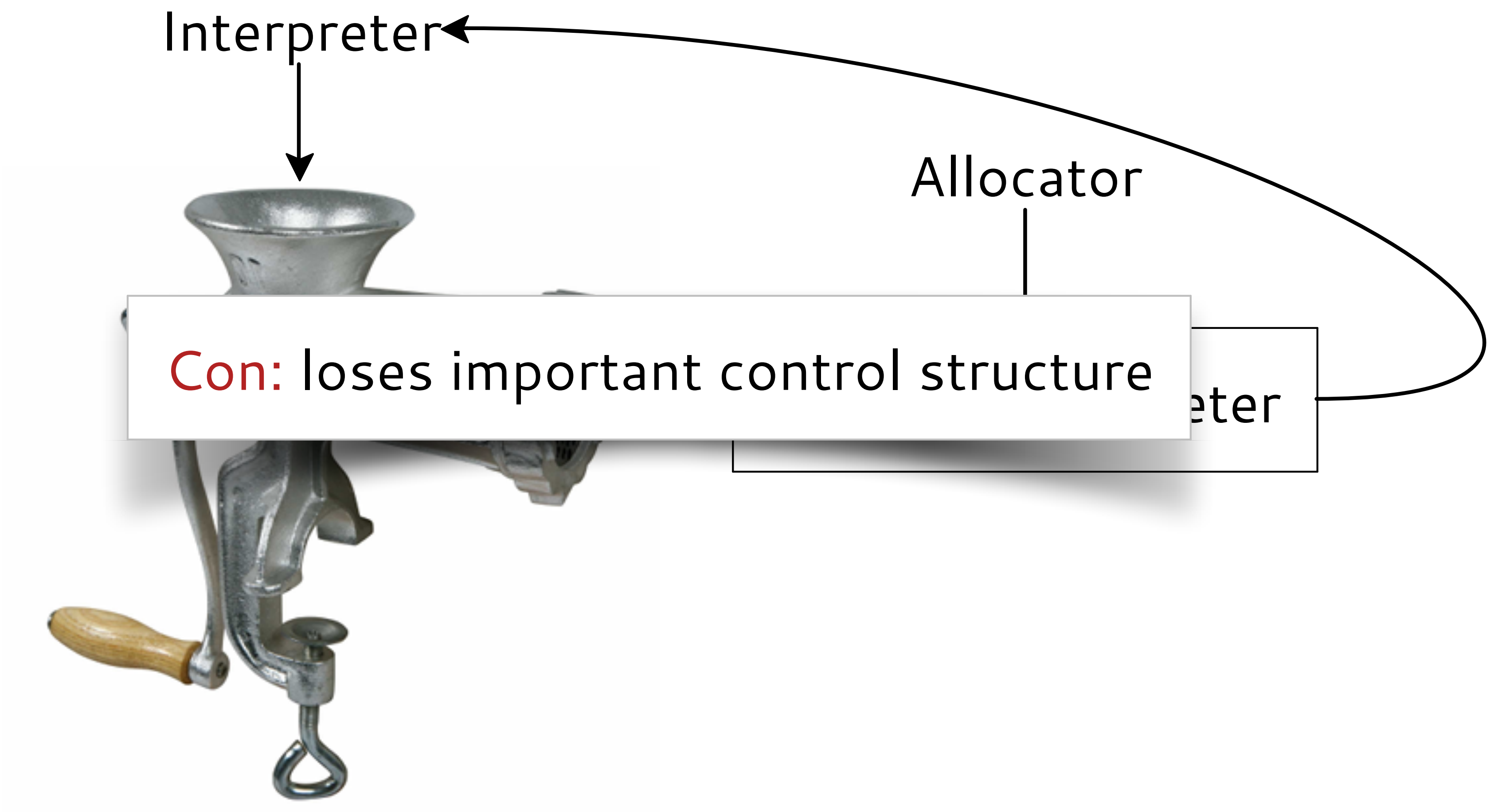
Allocator

Abstract interpreter

Abstracting Abstract Machines



Abstracting Abstract Machines



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?

akka

REST

Event

Clou

Actors

Operating systems

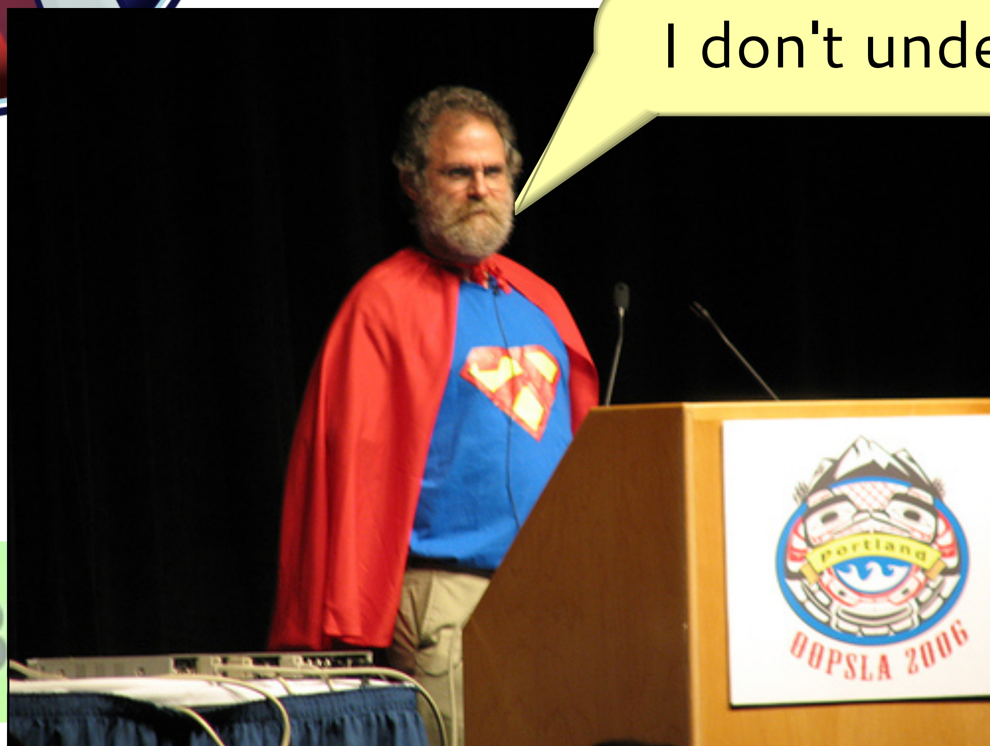
ines?)

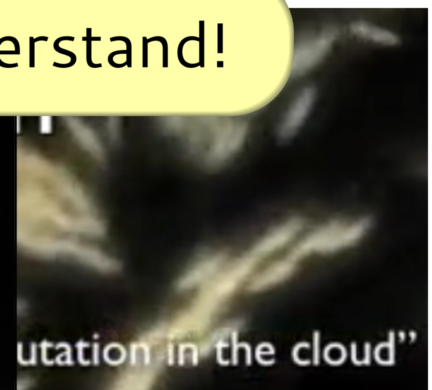
The Get Bonus
infinite entertainment system

Typesafe

Hekate — a highly-concurrent BitTorrent seeder.

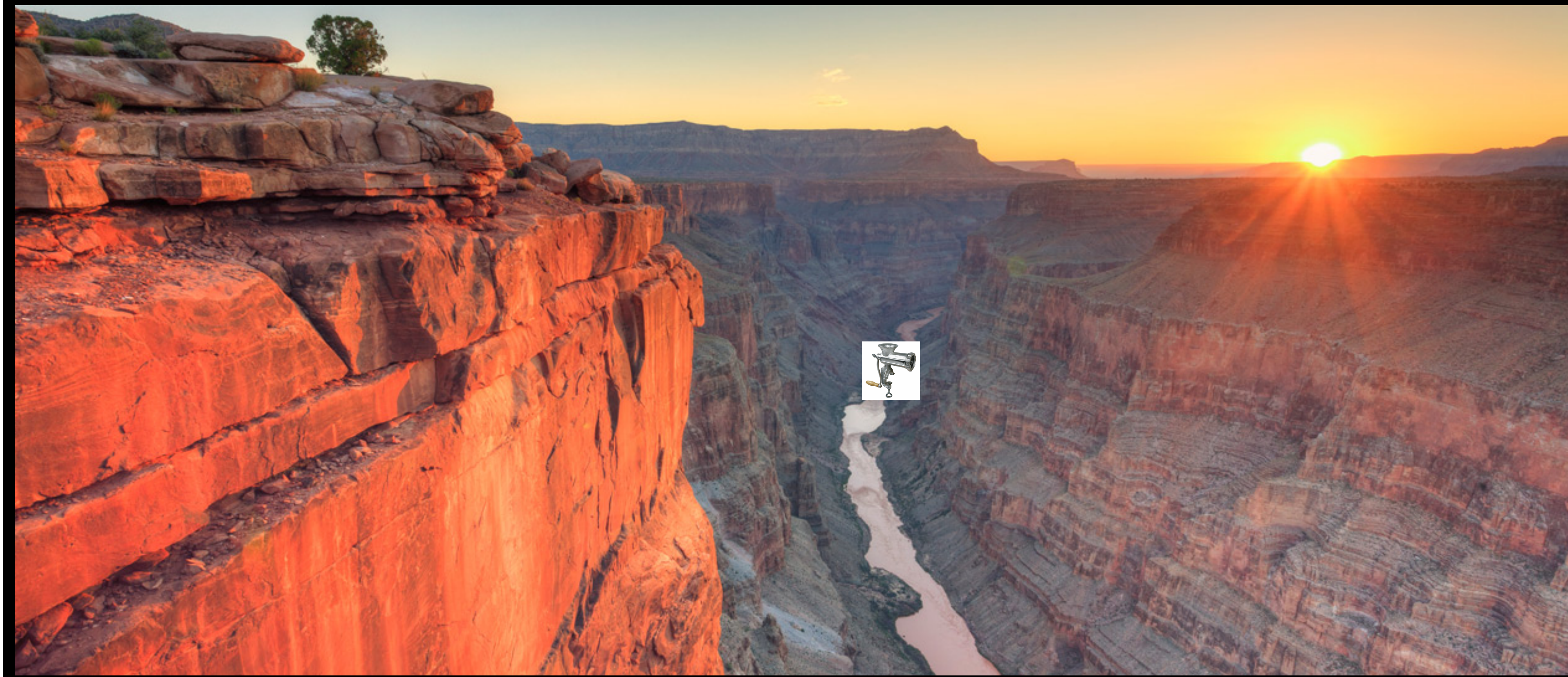
Who cares about continuations?





Hekate — a highly-concurrent BitTorrent seeder.

</motivation>



$$s \mapsto s'$$

$s \mapsto s'$  $\hat{s} \mathrel{\hat{\mapsto}} \hat{s}'$

Heap-allocate recursion

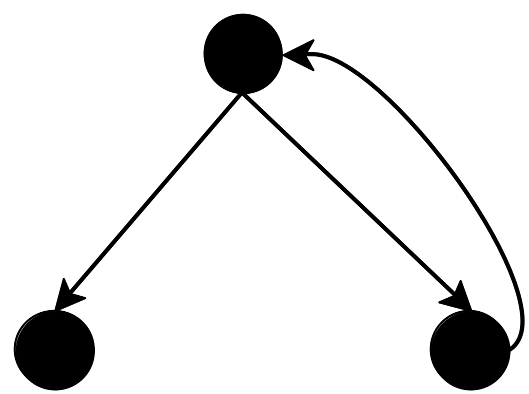
$\langle$code, heap, cont$\rangle$

$s \mapsto s'$  $\hat{s} \mathrel{\hat{\mapsto}} \hat{s}'$

Heap-allocate recursion

⟨code, heap, cont⟩

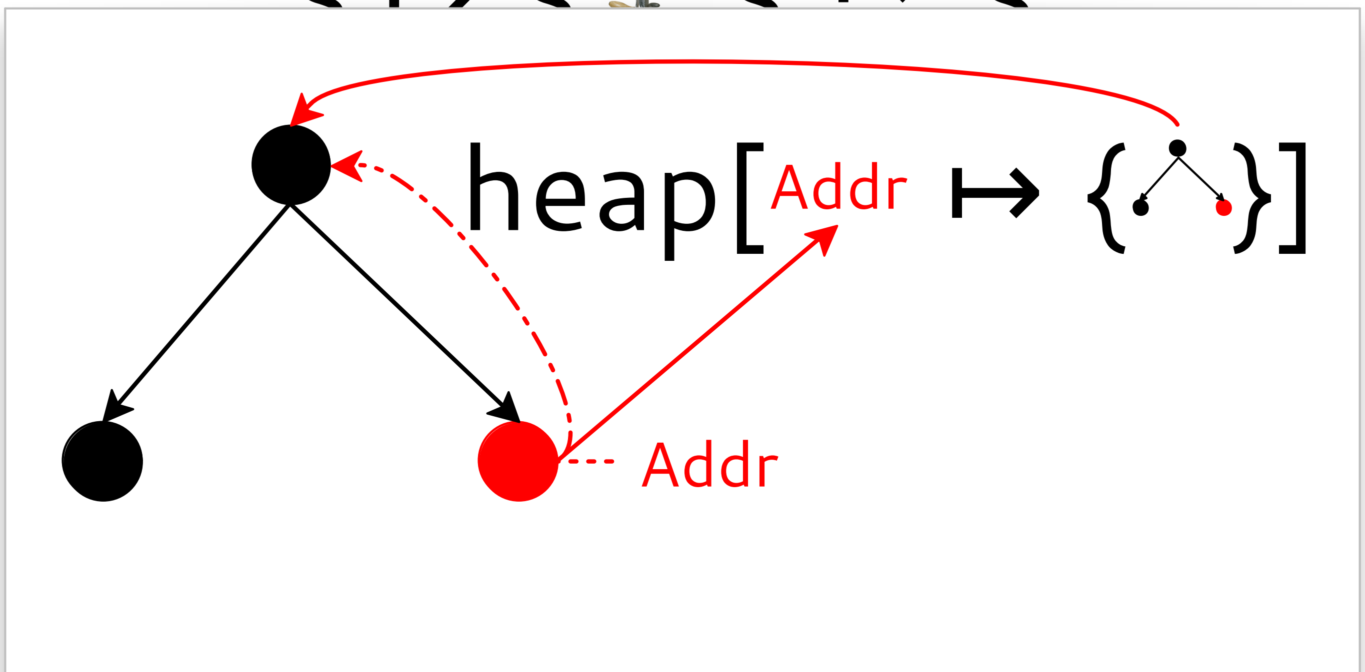
$s \mapsto s'$  $\hat{s} \;\hat{\mapsto}\; \hat{s}'$



Heap-allocate recursion

⟨code, heap, cont⟩

$s \mapsto s'$  $\hat{s} \mathrel{\hat{\mapsto}} \hat{s}'$



Heap-allocate recursion

$\langle$code, heap, cont$\rangle$

$s \mapsto s'$  $\hat{s} \mathrel{\hat{\mapsto}} \hat{s}'$

cont : List[Activation−Frame]

Heap-allocate recursion

$\langle$code, heap, cont$\rangle$

$s \mapsto s'$  $\hat{s} \mathrel{\hat{\mapsto}} \hat{s}'$

cont : List[Activation−Frame]

cons :X −> List[X] −> List[X]

Heap-allocate recursion

$\langle$code, heap, cont$\rangle$

$s \mapsto s'$  $\hat{s} \;\widehat{\mapsto}\; \hat{s}'$

cont : List[Activation−Frame]

cons :X -> Addr -> List[X]

Heap-allocate recursion

$\langle$code, heap, cont$\rangle$

$s \mapsto s'$  $\hat{s} \mathrel{\hat{\mapsto}} \hat{s}'$

cont : List[Activation−Frame]

cons :X −> Addr −> List[X]

heap : Map[Addr, Value]

Heap-allocate recursion

⟨code, heap, cont⟩

$s \mapsto s'$  $\hat{s} \mathrel{\hat{\mapsto}} \hat{s}'$

cont : List[Activation−Frame]

cons :X −> Addr −> List[X]

heap : Map[Addr, Set[Value]]

$h[a \mapsto v]$  $h[a \mapsto h(a) \cup \{v\}]$

Say we have some function `f : json -> html`

Say we have some function `f : json -> html`

We wrap it to validate its input and output

```
(λ (j)
   (if (good-json? j)
       (let ([r (f j)])
         (if (good-html? r)
             r
             (blame 'f)))
       (blame 'user)))
```

Say we have some function `f : json -> html`

We wrap it to validate its input and output

```
(λ (j)
   (if (good-json? j)
       (let ([r (f j)])
         (if (good-html? r)
             r
             (blame 'f)))
       (blame 'user)))
```

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

`read-request` blocks until json is read, then calls `f`

Say we have some function `f : json -> html`

We wrap it to validate its input and output

```
(λ (j)
   (if (good-json? j)
       (let ([r (f j)])
         (if (good-html? r)
             r
             (blame 'f)))
       (blame 'user)))

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

`read-request` blocks until json is read, then calls `f`

Say we have some function `f : json -> html`

We wrap it to validate its input and output

```
(λ (j)
   (if (good-json? j)
       (let ([r (f j)])
         (if (good-html? r)
             r
             (blame 'f)))
       (blame 'user)))

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

`read-request` blocks until json is read, then calls `f`

Say we have some function **`f : json -> html`**

We wrap it to validate its input and output

```
(λ (j)
   (if (good-json? j)
       (let ([r (f j)])
         (if (good-html? r)
             r
             (blame 'f)))
       (blame 'user)))

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

`read-request` blocks until json is read, then calls **`f`**

Say we have some function **f : json -> html**

We wrap it to validate its input and output

```
(λ (j)
   (if (good-json? j)
       (let ([r (f j)])
         (if (good-html? r)
             r
             (blame 'f)))
       (blame 'user)))

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

read-request blocks until json is read, then calls **f**

Say we have some function `f : json -> html`

We wrap it to validate its input and output

```
(λ (j)
   (if (good-json? j)
       (let ([r (f j)])
         (if (good-html? r)
             r
             (blame 'f)))
       (blame 'user)))

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

`read-request` blocks until json is read, then calls `f`

Say we have some function `f : json -> html`

We wrap it to validate its input and output

```
(λ (j)
   (if (good-json? j)
       (let ([r (f j)])
         (if (good-html? r)
             r
             (blame 'f)))
       (blame 'user)))

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

`read-request` blocks until json is read, then calls `f`

Say we have some function **`f : json -> html`**

We wrap it to validate its input and output

```
(λ (j)
   (if (good-json? j)
       (let ([r (f j)])
         (if (good-html? r)
             r
             (blame 'f)))
       (blame 'user)))
```

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

`read-request` blocks until json is read, then calls **`f`**

Insight:
delimit computations &
catalog contexts by revelant state

*The stack doesn't matter**

*yet

```
(λ (j)
   (if (good-json? j)
       (let ([r (f j)])
         (if (good-html? r)
             r
             (blame 'f)))
       (blame 'user)))

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

```
(λ (j)●
   (if (good-json? j)
       (let ([r (f j)])
         (if (good-html? r)
             r
             (blame 'f)))
       (blame 'user)))

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

Contexts = [● ↦ {cont}]

```
(λ (j)●●
   (if (good-json? j)
       (let ([r (f j)])
         (if (good-html? r)
             r
             (blame 'f)))
       (blame 'user)))

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

Contexts = [● ↦ {cont}, ● ↦ {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 `cons : X -> Addr -> List[X]`

Are ● just fancy addresses?

States are ⟨`code heap stack`⟩ and the stack is irrelevant

● is ⟨`code heap`⟩

What's really going on here?

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

Are ● just fancy addresses?

States are ⟨code heap stack⟩ and the stack is irrelevant

● is ⟨code heap⟩

h[⟨c,h'⟩ ↦ {cont}]

What's really going on here?

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

Are ● just fancy addresses?

States are ⟨`code heap stack`⟩ and the stack is irrelevant

● is ⟨`code heap`⟩

`h[⟨c,h'⟩ ↦ {cont}]`

What's really going on here?

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

Are ● just fancy addresses?

States are ⟨`code heap stack`⟩ and the stack is irrelevant

● is ⟨`code heap`⟩

`h[⟨c,h'⟩ ↦ {cont}]`

● are stored in a stratified heap: Contexts

What if "the stack" isn't a stack ?

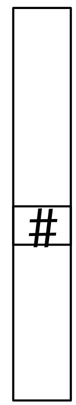
What if "the stack" isn't a stack ?

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

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

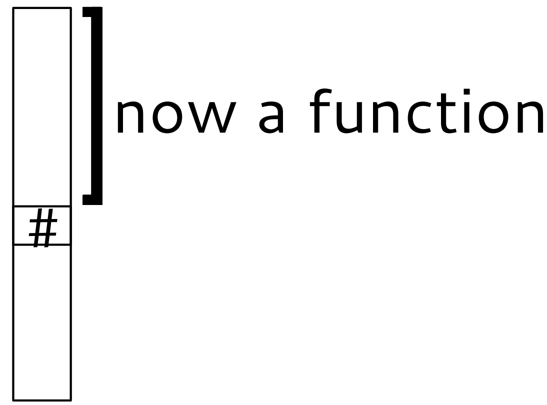
```
(+ 10 (reset (+ 2 (shift k (+ 40 (k (k 3)))))))
```

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



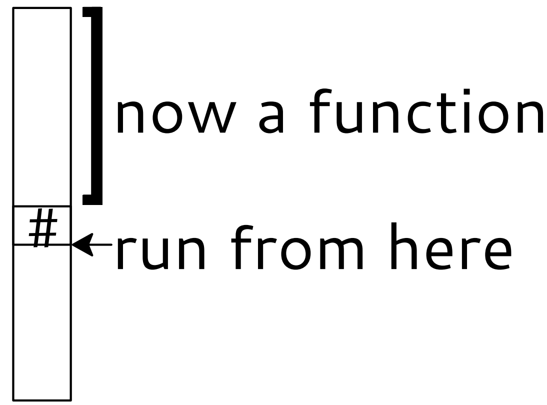

```
(+ 10 (reset (+ 2 (shift k (+ 40 (k (k 3)))))))
```

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



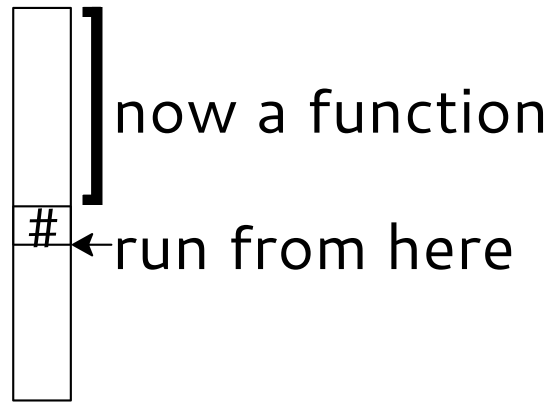

```
(+ 10 (reset (+ 2 (shift k (+ 40 (k (k 3)))))))
```

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




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

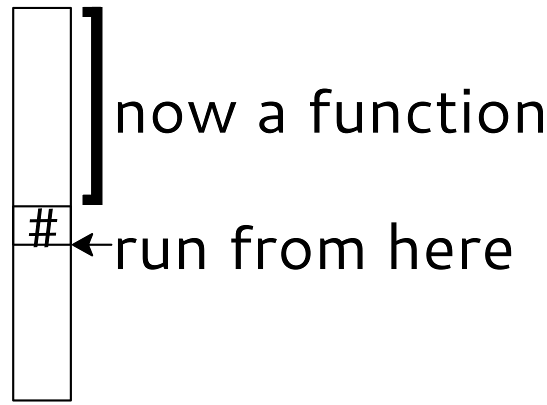
$$E[F[(\text{shift } k\ e)]] \mapsto E[e\{k := (\lambda\ (x)\ F[x])\}]$$




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

                  (+ 2 [])
```

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

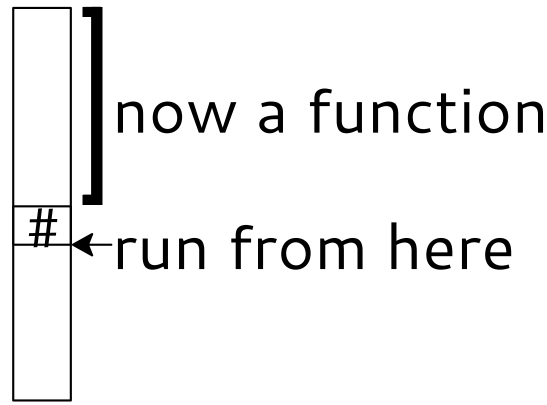



```
(+ 10 (reset (+ 2 (shift k (+ 40 (k (k 3)))))))

               (+ 2 [])

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

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

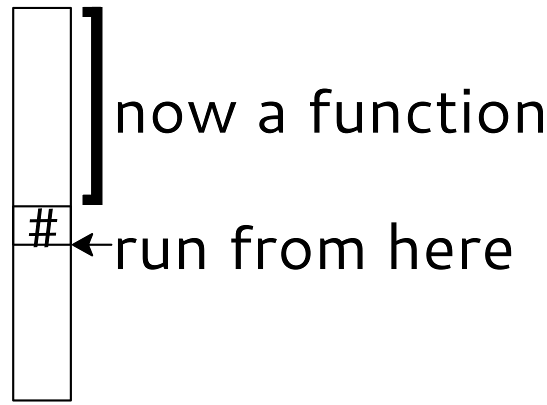



```
(+ 10 (reset (+ 2 (shift k (+ 40 (k (k 3)))))))

  k = (λ (x) (+ 2  x ))

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

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

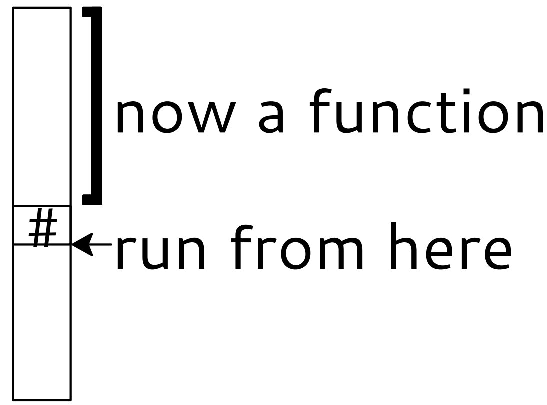



```
(+ 10 (reset (+ 2 (shift k (+ 40 (k (k 3)))))))

  k = (λ (x) (+ 2 x ))

(+ 10 (+ 40 ((λ (x) (+ 2 x)) ((λ (x) (+ 2 x)) 3)
```

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

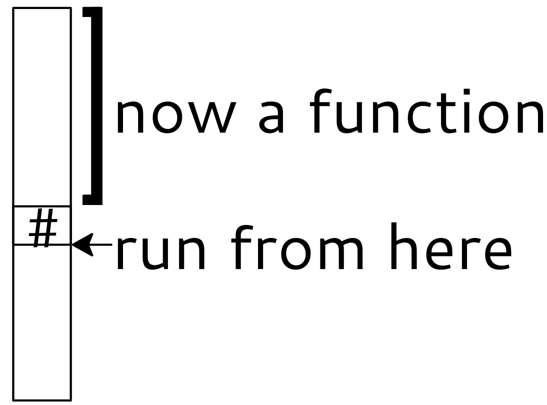



```
(+ 10 (reset (+ 2 (shift k (+ 40 (k (k 3)))))))

  k = (λ (x) (+ 2  x ))

(+ 10 (+ 40 (+ 2 (+ 2 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 = (λ (x) (+ 2  x ))

(+ 10 (+ 40 (+ 2 (+ 2 3))))

57
```

```
(λ (j)
  (if (good-json? j)
      (let ([r (f j)])
        (if (good-html? r)
            r
            (blame 'f)))
      (blame 'user)))

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

read-request uses non-blocking I/O

```
(λ (i)
```

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

```
         (blame 'f)))
      (blame 'user)))
```

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

read-request uses non-blocking I/O

```
(λ (i)
```

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

```
          (blame 'f)))
    (blame 'user)))
```

```
(document.w          request f)
                     request f)))
```

h[ka ↦ {(comp ●)}]

`read-request` uses non-blocking I/O

```
(λ (i)
```

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

```
          (blame 'f)))
     (blame 'user)))
```

```
(document.            quest f)
                      quest f)))
```

$h[ka \mapsto \{(comp\ \langle c,h'\rangle)\}]$

`read-request` uses non-blocking I/O

```
(λ (i)
```

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

```
      (blame 'f)))
    (blame 'user)))

(document. ... quest f)
               quest f)))
```

$h[ka \mapsto \{(comp\ \langle c,h'\rangle)\}]$

read-request uses non-blocking I/O

```
(define (read-request f)
  (shift k (evloop-until-evt
             (read-requ          f)
```

Can we stratify like with Contexts?

$h[ka \mapsto \{(comp\ \langle c,h'\rangle)\}]$

read-request uses non-blocking I/O

Of course not!

$\langle$ `(shift k e), heap,` ● $\rangle$ produces `heap(ka)` $\ni$ `(comp` $\langle$ `c,a` $\rangle$ `)`

Of course not!

$\langle$(shift k e), heap, ●$\rangle$ produces heap(ka) $\ni$ (comp $\langle$c,a$\rangle$)

χ(a) $\ni$ h'

Of course not!

$\langle$(shift k e), heap, ●$\rangle$ produces heap(ka) $\ni$ (comp $\langle$c,a$\rangle$)

χ(a) $\ni$ h'

Well, now χ is relevant!

Of course not!

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

$$\chi(\texttt{a}) \ni \texttt{h'}$$

Well, now χ is relevant! Since χ closes the heap

Of course not!

$\langle$`(shift k e), heap,` ● $\rangle$ produces `heap(ka)` $\ni$ `(comp` $\langle$`c,a`$\rangle$`)`

$$\chi(a) \ni h'$$

Well, now χ is relevant! Since χ closes the heap

● $\equiv$ $\langle$`c', h', χ'`$\rangle$

Of course not!

⟨(shift k e), heap, ●⟩ produces heap(ka) ∋ (comp ⟨c,a⟩)

χ(a) ∋ h'

Well, now χ is relevant! Since χ closes the heap

● ≡ ⟨c', h', χ'⟩

χ(a) ∋ ⟨h',χ'⟩

Of course not!

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

$$\chi(\texttt{a}) \ni \texttt{h'}$$

Well, now χ is relevant! Since χ closes the heap

$$\bullet \equiv \langle \texttt{c'},\ \texttt{h'},\ \chi' \rangle$$

$$\chi(\texttt{a}) \ni \langle \texttt{h'}, \chi' \rangle$$

Of course not!

$\langle$`(shift k e), heap,` ● $\rangle$ produces heap(ka) $\ni$ `(comp` $\langle$`c,`$a\rangle$`)`

$$\chi(a) \ni h'$$

Well,

χ and heap are mutually recursive! Can't stratify!

● $\equiv \langle c, h, \chi \rangle$

$$\chi(a) \ni \langle h', \chi' \rangle$$

Squash it

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

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

$[\![\langle c', a \rangle]\!] = \{cont \in \text{Contexts}(\langle c', h', \chi' \rangle) : h' \in \chi(a), \chi' \sqsubseteq \chi\}$

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

$h = []$

$\chi = []$

```
(λ
        (if (good-html? f)
            r
            (blame 'f)))
    (b
(document                          uest f)
                                   uest f)))
```

```
(define (read-request f)●
  (shift k (evloop-until-evt
             (read-request-evt f)
             k)))
```

```
(λ
          r
          (blame 'f)))
      (b
(document                              uest f)
                                       uest f)))
```

h = [ka ↦ {(comp ⟨◖,a⟩)}]

χ = ◗ ⊔ [a ↦ {◥}]

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

$$h = [ka \mapsto \{(\mathrm{comp}\ \langle\ ,a\rangle), (\mathrm{comp}\ \langle\ ,a\rangle)\}]$$

$$\chi = \sqcup \sqcup [a \mapsto \{ , \}]$$

```
(λ
  r
  (blame 'f)))
(doc                                  f)
                                      f)))
```

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

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

$$h = [\mathrm{ka} \mapsto \{(\mathrm{comp}\ \langle\ ,\mathrm{a}\rangle)\},\quad \mathrm{ka} \mapsto \{(\mathrm{comp}\ \langle\ ,\mathrm{a}\rangle)\}]$$

$$\chi = \ \sqcup\ \sqcup [\mathrm{a} \mapsto \{\ \}] \sqcup [\mathrm{a} \mapsto \{\ \}]$$

```
(docu
      t f)
      t f)))
```

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

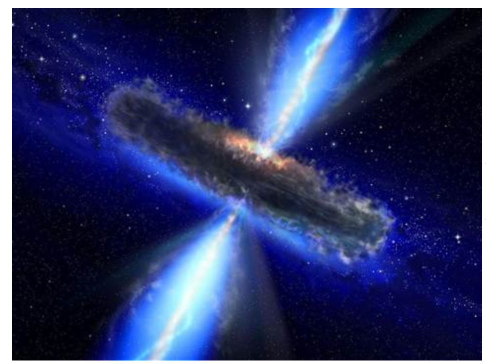
Want shift/reset in modular semantics

(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

Takeaway

Delimit computations by relevant state

Squash abstracted relevance objects

Break cycles in state space with addresses

Thank you