Y Combinator, Context and Redex, and Continuations

Kristopher Micinski (Slides in part by Thomas Gilray, UAB)

CIS 700 —Program Analysis: Foundations and Applications

Fall ’19, Syracuse University

Leaving off last class…

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
x ::= <vars>
```

Dechurching is easy to define and illuminative to how the encodings work.

```
(define (church->nat cv)
        ((cv add1) 0))


(define (church->list cv)
        ((cv (λ (car)
               (λ (cdr)
                 (cons car
                   (church->list cdr)))))
         (λ (na) ‘())))


(define (church->bool cv)
        ((cv (λ () #t))
         (λ () #f)))
```

```
(define U (λ (f) (f f)))
```

```
(letrec ([fib (lambda (x) (if (= x 0) 1 (* x (fib (- x 1)))))])
  (fib 3))
```

```
(let ([fib (U (lambda (f)
                (lambda (x) (if (= x 0) 1 (* x (... (- x 1)))))))])
  (fib 3))
```

(f f)

To translate letrec -> U, formula is…

- Translate letrec to applications of U
- Pass (lambda (foo) …) to U
- Change recursive calls to (foo foo)

```
(letrec ([fib (lambda (x)
                (if (= x 0)
                    1
                   (* x (fib (- x 1)))))])
  (fib 3))
```

```
(let ([fib (U (lambda (f)
              (lambda (x) (if (= x 0) 1 (* x ((f f) (- x 1)))))))])
  (fib 3))
```

Consider as we evaluate (U (lambda (f) …))

First, we evaluate U to a closure

```
(define U (lambda (g) (g g)))

(let ([fib (U (lambda (f)
                (lambda (x)
                   (if (= x 0)
                       1
                       (* x ((f f) (- x 1)))))))])
  (fib 3))
```

Next, we evaluate (lambda (f) …) to a closure, which is stored on the heap at some address..

addr

(lambda (f) …), env

```
(define U (lambda (g) (g g))

(let ([fib (U (lambda (f)
                (lambda (x)
                   (if (= x 0)
                       1
                       (* x ((f f) (- x 1)))))))])
  (fib 3))
```

Then evaluate application of U to (lambda (f) …) and jump **here**, but in environment where g points at **this**

addr (lambda (f) …), env

```
(define U (lambda (g) (g g)))

(let ([fib (U (lambda (f)
                (lambda (x)
                   (if (= x 0)
                       1
                       (* x ((f f) (- x 1)))))))])
  (fib 3))
```

After invoking g (which is (lambda (f) …)) on itself **here**,
we **return** a closure for (lambda (x) …)

addr

(lambda (f) …), env

```
(define U (lambda (g) (g g)))

(let ([fib (U (lambda (f)
                (lambda (x)
                   (if (= x 0)
                       1
                       (* x ((f f) (- x 1)))))))])
  (fib 3))
```

After invoking g (which is (lambda (f) …)) on itself **here**, we **return** a closure for (lambda (x) …)

Notice that that closure binds f to itself (via addr).

addr

(lambda (f) …), env

addr’

(lambda (x) …),
f |-> addr

```
(define U (lambda (g) (g g))

(let ([fib (U (lambda (f)
                (lambda (x)
                   (if (= x 0)
                       1
                       (* x ((f f) (- x 1)))))))])
  (fib 3))
```

The evaluation of (f f) will produce a **new** (lambda (x) …), which **also** binds f

addr”

(lambda (x) …),
f |-> addr

addr

(lambda (f) …), env

addr’

(lambda (x) …),
f |-> addr

```
(define U (lambda (g) (g g)))

(let ([fib (U (lambda (f)
               (lambda (x)
                  (if (= x 0)
                      1
                      (* x ((f f) (- x 1)))))))])
  (fib 3))
```

Now when this new closure is invoked on (- x 1), the whole process starts over…

addr”

(lambda (x) …), f |-> addr

addr

(lambda (f) …), env

addr’

(lambda (x) …), f |-> addr

```
(define U (lambda (g) (g g))

(let ([fib (U (lambda (f)
               (lambda (x)
                 (if (= x 0)
                     1
                     (* x ((f f) (- x 1)))))))])
  (fib 3))
```

At an ultra-high level the U combinator is saying…

Can simulate recursion via **generator** function
Generator will copy of itself **on the fly**

```
(define U (lambda (g) (g g))

(let ([fib (U (lambda (f)
                (lambda (x)
                   (if (= x 0)
                       1
                       (* x ((f f) (- x 1)))))))])
  (fib 3))
```

U combinator isn't **ideal**
(i.e., want to use built-in recursion if possible)

(Notice: generates new closure to simulate each recursive call!)

addr” (lambda (x) …), f |-> addr

addr (lambda (f) …), env

addr’ (lambda (x) …), f |-> addr

```
(define U (lambda (g) (g g)))

(let ([fib (U (lambda (f)
                (lambda (x)
                  (if (= x 0)
                      1
                      (* x ((f f) (- x 1)))))))])
  (fib 3))
```

Y combinator

Any lambda calculus term that satisfies…

$$(Y\ f) = f\ (Y\ f)$$

(Not unique, there are different Y combinators for, e.g., Call-by-value vs. name)

A fixed point of a function is value mapped to itself by that function.

This is an ultra-broad definition, and it is not at first clear why it's useful in implementing recursion.

Y (lamba (f) …) = f (Y (lambda (f) …)

Using this equivalence…

fib = (Y (lambda (f) …)

 = (f (Y (lambda (f) …)))

 = (f (f (Y (lambda (f) …))))

 = (f (f (f (Y (lambda (f) …)))))

```
(let ([fib (Y (lambda (f)
                (lambda (x)
                   (if (= x 0)
                       1
                       (* x (f (- x 1)))))))])
  (fib 3))
```

Three step process for deriving Y

```
(Y f) = f (Y f)
```

```
Y = (λ (f) (f (Y f)))
```

1. Treat as definition

```
mY = (λ (mY)
       (λ (f)
         (f ((mY mY) f))))
```

2. Lift to mk-Y, use self-application

```
mY = (λ (mY)
       (λ (f)
         (f (λ (x) (((mY mY) f) x)))))
```

3. Eta-expand

U-combinator: `(U U)` **is Omega**

```
Y  =  (U (λ (y) (λ (f)
           (f (λ (x) (((y y) f) x))))))
```

```
(let ([fact (Y (λ (fact) (λ (n)
                 (if (= n 0)
                     1
                     (* n (fact (- n 1))))))])
  (fact 5))
```

Example

```
(define Y ((λ (x) (x x)) (λ (y) (λ (f)
                                          (f (λ (x) (((y y) f) x)))))))
```

```
(define (fib x)
  (if (or (= x 0) (= x 1))
      1
      (+ (fib (- x 1)) (fib (- x 2)))))
```

Rewrite this to use the Y combinator instead

Evaluation contexts

Restrict the order in which we may simplify a program's redexes

$$
\begin{aligned}
\mathcal{E} \; ::= \; & (\mathcal{E}\ \ e) \\
\mid \; & (v\ \ \mathcal{E}) \\
\mid \; & \square
\end{aligned}
$$

(left-to-right) CBV evaluation

$$
\begin{aligned}
\mathcal{E} \; ::= \; & (\mathcal{E}\ \ e) \\
\mid \; & \square
\end{aligned}
$$

(left-to-right) CBN evaluation

$$
v \; ::= \; (\lambda\ (x)\ e)
$$

$$
\begin{aligned}
e \; ::= \; & (\lambda\ (x)\ e) \\
\mid \; & (e\ \ e) \\
\mid \; & x
\end{aligned}
$$

Context and redex

For CBV a redex must be `(v v)`
For CVN, a redex must be `(v e)`

$$\mathcal{E}[\,\overbrace{\texttt{(v v)}}^{\texttt{r}}\,] =$$

`(((λ (x) ((λ (y) y) x)) (λ (z) z)) (λ (w) w))`

$\mathcal{E}$ = `(□ (λ (w) w))`

`r = ((λ (x) ((λ (y) y) x)) (λ (z) z))`

Context and redex

$\mathcal{E}$[r] =

(((λ (x) ((λ (y) y) x)) (λ (z) z)) (λ (w) w))

$\mathcal{E}$ = (□ (λ (w) w))

r = ((λ (x) ((λ (y) y) x)) (λ (z) z))

$\rightarrow_\beta$ ((λ (y) y) (λ (z) z))

Put the reduced redex back in its evaluation context…

$\mathscr{E}$ = `(□ (λ (w) w))`

r = `((λ (x) ((λ (y) y) x)) (λ (z) z))`

$\to_\beta$ `((λ (y) y) (λ (z) z))`

$\Big\downarrow$ $\mathscr{E}$[r]

`(((λ (y) y) (λ (z) z)) (λ (w) w))`

Exercises — can you evaluate…

1) `(((λ (y) y) (λ (z) z)) (λ (w) w))`

2) `((λ (u) (u u)) (λ (x) (λ (x) x)))`

3)
```
(((λ (x) x) (λ (y) y))
 ((λ (u) (u u)) (λ (z) (z z))))
```

Continuations: first-class control

Continuations

A ***continuation*** is a return point, a call stack, or the remainder of the program, viewed as a function.

In Scheme, continuations are first-class values that can be captured using the language form `call/cc` and passed around to be invoked later.

First-class continuations

We may consider several alternative viewpoints on first-class continuations:

A ***continuation*** is a value encoding a *saved return point* to resume.

A ***continuation*** is a function encoding the *remainder of the program*.

A ***continuation*** is a function that never returns. When invoked on an input value, it resumes a previous return point with that value, and finishes the program from that return point until it exits.

Continuations generalize all known control constructs: gotos, loops, return statements, exceptions, C's `longjmp`, threads/coroutines, etc

Continuations

Continuations are said to permit ***time travel***, in the sense that they permit jumping back to a saved dynamic *evaluation context*: a *previous* call stack, or point in time, or a *future* one!

This may shorten the stack

…grow the stack

…or replace it entirely.

`(call/cc e`$_0$`)`

call with current continuation

`call/cc` takes a single argument, a callback, which it applies on the ***current continuation***—that is, the return from call/cc as a first-class function that saves the full call stack under call/cc.

```
(+ 1 (call/cc (lambda (k) (k 2))))
;; => 3
```

Takes the call stack at the second argument expression of `(+ ...)` and saves it, essentially as a function, bound to `k`, that can be invoked on a value for that expression at a later point in time.

When k is invoked on the number `2`, execution jumps back to the saved return point for `call/cc` and returns `2`, returning `3` from the program as a whole.

```
(+ 1 (call/cc (lambda (k) (k 2))))
;; => 3
```

The program never returns from call `(k 2)` because ***undelimited continuations*** run until the program exits.

`call/cc` gives us undelimited (a.k.a. full) continuations.

```
(+ 1 (call/cc (lambda (k) (k 2) (print 0))))
;; => 3       (print 0) is never reached
```

```
(+ 1 (call/cc (lambda (k) (k 2))))
;; => 3
```

This `call/cc`'s behavior is *roughly* the same as the application:

```
((lambda (k) (k 2))
 (lambda (n) (exit (print (+ 1 n)))))
;; => 3
```

Where the high-lit continuation `(lambda (n) …)` takes a return value for the `(call/cc …)` expression and finishes the program.

```
(let ([cc (call/cc (lambda (k) k))])
    ...)
```

A common idiom for `call/cc` is to let-bind the current continuation.

```
(let ([cc (call/cc (lambda (k) k))])
  ...)
```

Note that applying call/cc on the identity function is exactly the same as applying it on the u-combinator!

```
(let ([cc (call/cc (lambda (k) (k k)))])
  ...)
```

Why is this the case?

`call/cc` makes a tail call to `(lambda (k) …)`, so the body of the function is the same return point as the captured continuation k!

```
(let ([cc (call/cc (lambda (k) k))])
  ...)
```

This return point ...is the same as this one…

```
(let ([cc (call/cc (lambda (k) (k k)))])
  ...)
```

…and calling k on itself, returns k to itself!

Returning value v is the same as *calling* that saved return point *on* v.

```
(let ([cc (call/cc (lambda (k) k))])

  ;; loop body goes here

  (if (jump-to-top?)
      (cc cc)
      return-value))
```

Continuations can be used to jump back to a previous point.

Just as we could have invoked `call/cc` on the u-combinator, to jump back to the let-binding of `cc`, returning `cc`, we call `(cc cc)`.

```
(define (fun x)

  (let ([y (if (p? x)
                 …
                 …)])

    (g x y)))
```

A simple use of continuations is to implement a ***preemptive return***.

What if we wanted to return from `fun` within the right-hand-side of the `let` form?

Binds the return-point of the current call to `fun` to a continuation `return`.

```
(define (fun x)
  (call/cc (lambda (return)

    (let ([y (if (p? x)
                 …
                 (return x))])

      (g x y)))))
```

Uses the continuation `return` to jump back to the return point of `fun` and yield value `x` instead of binding `y` and calling `g`.

Try an example. What do each of these 3 examples return?
(Hint: Racket evaluates argument expressions left to right.)

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 3)))))))
```

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 (k1 3))))))))
```

```
(call/cc (lambda (k0)
           (+ 1
              (call/cc (lambda (k1)
                         (+ 1 (k1 3))))
              (k0 1))))
```

Try an example. What do each of these 3 examples return?
(Hint: Racket evaluates argument expressions left to right.)

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                            (+ 1 (k0 3)))))))
```

3

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                            (+ 1 (k0 (k1 3))))))))
```

4

```
(call/cc (lambda (k0)
           (+ 1
              (call/cc (lambda (k1)
                          (+ 1 (k1 3))))
              (k0 1))))
```

1

Continue and break

A Python `while` loop on the left that supports `continue` and `break` can be implemented using `call/cc` as the Scheme on the right.

```
while cond:
  body
else:
  otherwise
```

```
(call/cc (λ (break)
  (letrec ([loop (λ ()
                   (when cond
                     (call/cc (λ (continue)
                                body))
                     (loop)))])
    (loop)
    otherwise)))
```

Continuations and mutation

```
(let* ([n 2]
       [cc (call/cc (lambda (k) k))])
  (set! n (+ n 1))
  (if (<= n 4)
      (cc cc)
      n))
```

Does this program terminate? What does it return?

Continuations and mutation

```
(let* ([n 2]
       [cc (call/cc (lambda (k) k))])
  (set! n (+ n 1))
  (if (<= n 4)
      (cc cc)
      n))
```

This loop terminates and returns 5.

This illustrates that invoking a continuation resumes a previous call stack, but *does not* revert mutations—changes made in the heap.

Try an example. What do each of these 2 examples return?
(Hint: Racket evaluates argument expressions left to right.)

```
(define n 3)
(+ n (call/cc
       (lambda (cc)
         (set! n (+ n 1))
         (cc 1))))
```

```
(define n 3)
(+ (call/cc
       (lambda (cc)
         (set! n (+ n 1))
         (cc 1)))
   n)
```

Try an example. What do each of these 2 examples return?
(Hint: Racket evaluates argument expressions left to right.)

```
(define n 3)
(+ n (call/cc
       (lambda (cc)
         (set! n (+ n 1))
         (cc 1))))
```

4

```
(define n 3)
(+ (call/cc
       (lambda (cc)
         (set! n (+ n 1))
         (cc 1)))
   n)
```

5

Stack-passing (CEK) semantics

(implementing first-class continuations)

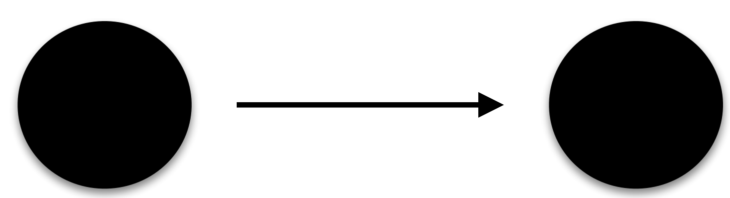
C	**Control-expression**	Term-rewriting / textual reduction Context and redex for deterministic eval
CE	**Control & Env machine**	Big-step, explicit closure creation
CES	**Store-passing machine**	Passes addr->value map in evaluation order
CEK	**Stack-passing machine**	Passes a list of stack frames, small-step

$$\frac{(\mathtt{e}_0, \text{env}) \Downarrow (\mathtt{(\lambda\ (x)\ e_2)}, \text{env}') \qquad (\mathtt{e}_1, \text{env}) \Downarrow v_1 \qquad (\mathtt{e}_2, \text{env}'[\mathtt{x} \mapsto v_1]) \Downarrow v_2}{(\mathtt{(e_0\ e_1)}, \text{env}) \Downarrow v_2}$$

$$\frac{}{(\mathtt{(\lambda\ (x)\ e)}, \text{env}) \Downarrow (\mathtt{(\lambda\ (x)\ e)}, \text{env})}$$

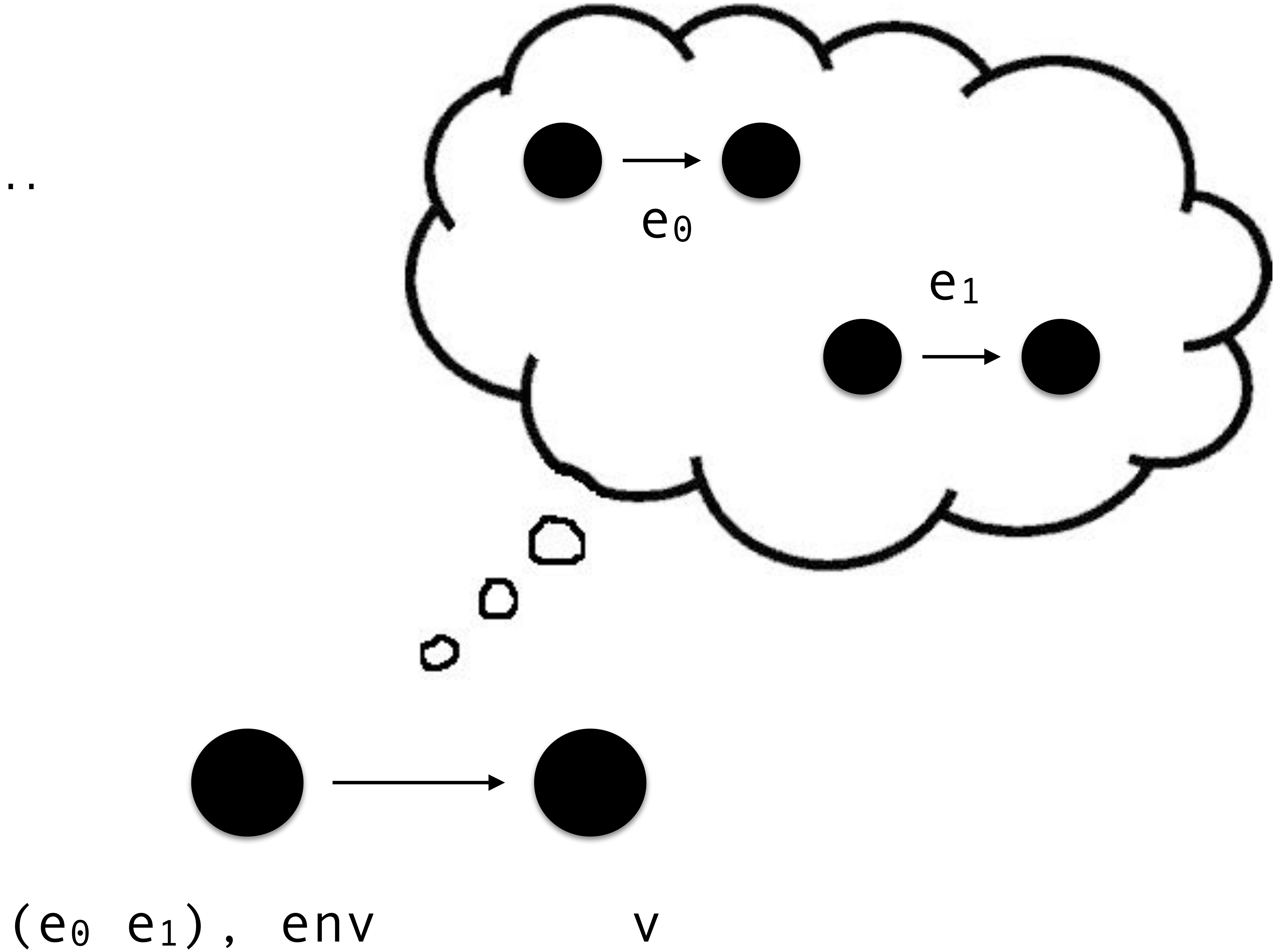
$$\frac{}{(\mathtt{x}, \text{env}) \Downarrow \text{env}(\mathtt{x})}$$

Previously…



$(e_0\ e_1),\ env$ $\qquad$ v

Previously…



```
(define (interp e env)
  (match e
         [(? symbol? x)
          (hash-ref env x)]

         [`(λ (,x) ,e0)
          `(clo (λ (,x) ,e0) ,env)]

         [`(,e0 ,e1)
          (define v0 (interp e0 env))
          (define v1 (interp e1 env))
          (match v0
            [`(clo (λ (,x) ,e2) ,env)
             (interp e2 (hash-set env x v1))])]))
```

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

k ::= **halt** | **ar**(e, env, k)
| **fn**(v, k)

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

k ::= **halt** | **ar**(e, env, k)
| **fn**(v, k)

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

$$\mathscr{E} ::= (\mathscr{E}\ \ e) \mid (v\ \ \mathscr{E}) \mid \square$$

$$((\texttt{e}_0\ \texttt{e}_1), env, k) \rightarrow (\texttt{e}_0, env, \mathbf{ar}(\texttt{e}_1, env, k))$$

$$(\texttt{x}, env, \mathbf{ar}(\texttt{e}_1, env_1, k_1)) \rightarrow (\texttt{e}_1, env_1, \mathbf{fn}(env(\texttt{x}), k_1))$$

$$((\lambda\ (\texttt{x})\ \texttt{e}), env, \mathbf{ar}(\texttt{e}_1, env_1, k_1)) \rightarrow (\texttt{e}_1, env_1, \mathbf{fn}(((\lambda\ (\texttt{x})\ \texttt{e}), env), k_1))$$

$$(\texttt{x}, env, \mathbf{fn}(((\lambda\ (\texttt{x}_1)\ \texttt{e}_1), env_1), k_1)) \rightarrow (\texttt{e}_1, env_1[\texttt{x}_1 \mapsto env(\texttt{x})], k_1)$$

$$((\lambda\ (\texttt{x})\ \texttt{e}), env, \mathbf{fn}(((\lambda\ (\texttt{x}_1)\ \texttt{e}_1), env_1), k_1)) \rightarrow (\texttt{e}_1, env_1[\texttt{x}_1 \mapsto ((\lambda\ (\texttt{x})\ \texttt{e}), env)], k_1)$$

call/cc semantics

$$(\texttt{(call/cc (}\lambda\texttt{ (x) e}_0\texttt{))}, \text{env}, k) \rightarrow (\texttt{e}_0, \text{env}[\texttt{x} \mapsto k], k)$$

$$(\texttt{(}\lambda\texttt{ (x) e}_0\texttt{)}, \text{env}, \textbf{fn}(k_0, k_1)) \rightarrow (\texttt{(}\lambda\texttt{ (x) e}_0\texttt{)}, \text{env}, k_0)$$

$$(\texttt{x}, \text{env}, \textbf{fn}(k_0, k_1)) \rightarrow (\texttt{x}, \text{env}, k_0)$$

$$e ::= \ldots \mid (\texttt{let}\ ([x\ e_0])\ e_1)$$

$$k ::= \ldots \mid \textbf{let}(x, e, env, k)$$

$$(x, env, \textbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$$

$$((\lambda\ (x)\ e), env, \textbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto ((\lambda\ (x)\ e), env)], k_1)$$

$$e ::= \ldots \mid (\texttt{let} \ ([x \ e_0]) \ e_1)$$

$$(x, env, \mathbf{fn}((\texttt{(}\lambda \ (x_1) \ e_1\texttt{)}, env_1), k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$$

$$(\texttt{(}\lambda \ (x) \ e\texttt{)}, env, \mathbf{fn}((\texttt{(}\lambda \ (x_1) \ e_1\texttt{)}, env_1), k_1)) \rightarrow (e_1, env_1[x_1 \mapsto (\texttt{(}\lambda \ (x) \ e\texttt{)}, env)], k_1)$$

$$k ::= \ldots \mid \mathbf{let}(x, e, env, k)$$

These are nearly identical because a let form is just an immediate application of a lambda!

$$(x, env, \mathbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$$

$$(\texttt{(}\lambda \ (x) \ e\texttt{)}, env, \mathbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto (\texttt{(}\lambda \ (x) \ e\texttt{)}, env)], k_1)$$

CEK-machine evaluation

$$
\begin{aligned}
(e_0, [], ()) &\to \ldots \\
&\to \ldots \\
&\to \ldots \\
&\to \ldots \\
&\to (x, env, \textbf{halt}) \to env(x)
\end{aligned}
$$

consider the following question.

Is it possible to take an arbitrary Racket/Scheme program and transform it systematically so that no function ever returns?