# Continuations

CIS400 (Compiler Construction)

Kris Micinski, Fall 2021

# 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 e0)`

## 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.

```scheme
(+ 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**

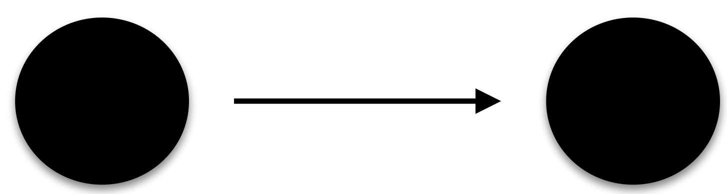
Passes a list of stack frames, small-step

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

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

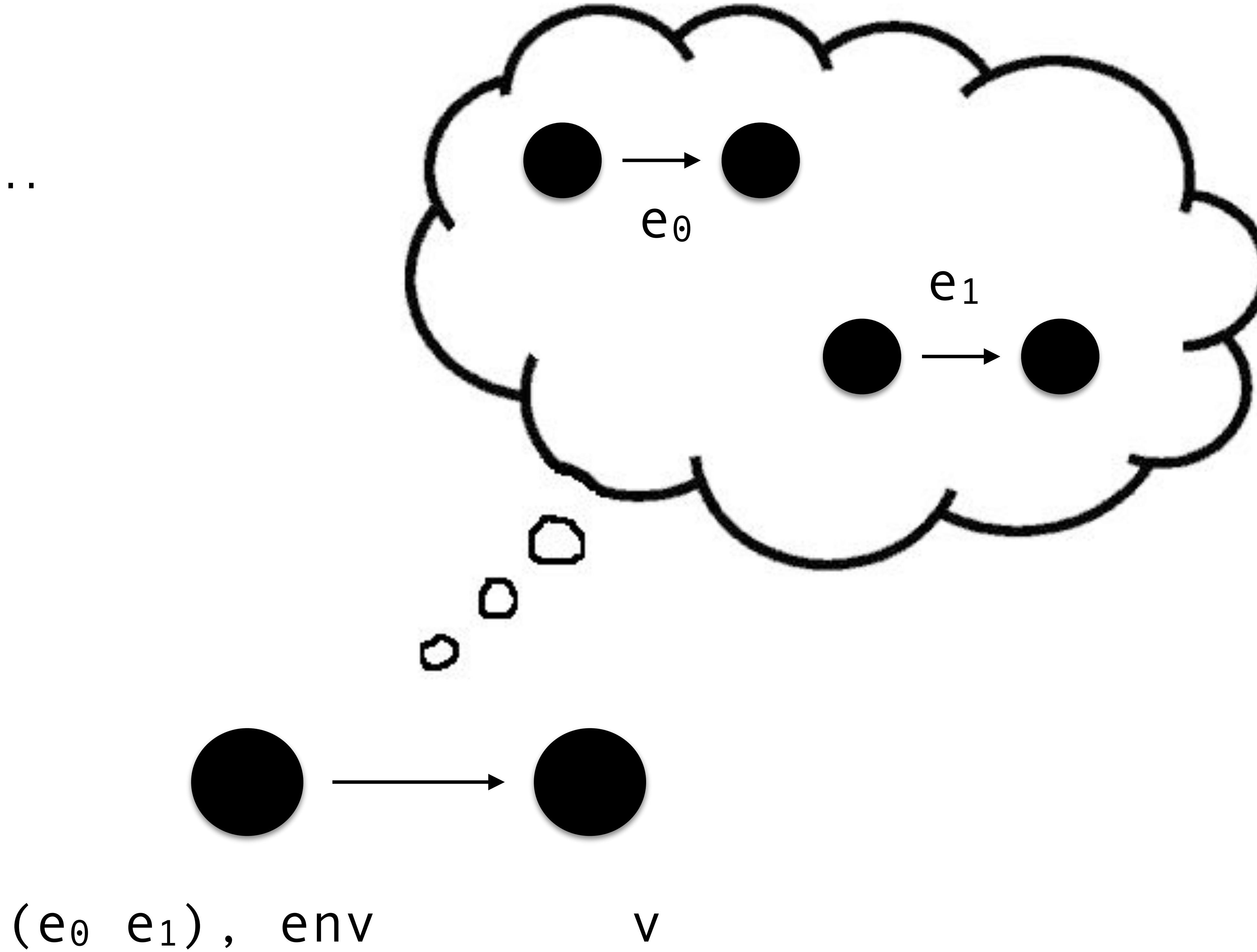
$$\frac{}{(x, \mathrm{env}) \Downarrow \mathrm{env}(x)}$$

# 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))
```

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

$$((e_0\ e_1), env, k) \rightarrow (e_0, env, \mathbf{ar}(e_1, env, k))$$

$$(x, env, \mathbf{ar}(e_1, env_1, k_1)) \rightarrow (e_1, env_1, \mathbf{fn}(env(x), k_1))$$

$$((\lambda\ (x)\ e), env, \mathbf{ar}(e_1, env_1, k_1)) \rightarrow (e_1, env_1, \mathbf{fn}(((\lambda\ (x)\ e), env), k_1))$$

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

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

# call/cc semantics

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

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

$$(\texttt{x}, \text{env}, \mathbf{fn}(k_0, k_1)) \to (\texttt{x}, \text{env}, k_0)$$

$$\texttt{e ::= ... | (let ([x e}_0\texttt{]) e}_1\texttt{)}$$

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

$$(\texttt{x}, env, \textbf{let}(x_1, \texttt{e}_1, env_1, k_1)) \ \rightarrow \ (\texttt{e}_1, env_1[x_1 \mapsto env(x)], k_1)$$

$$(\texttt{(}\lambda\texttt{ (x) e)}, env, \textbf{let}(x_1, \texttt{e}_1, env_1, k_1)) \ \rightarrow \ (\texttt{e}_1, env_1[x_1 \mapsto (\texttt{(}\lambda\texttt{ (x) e)}, env)], k_1)$$

$\mathtt{e}$ ::= ...

$\quad | \ \mathtt{(let\ ([x\ e_0])\ e_1)}$

$$(\mathtt{x}, env, \textbf{fn}((\mathtt{(\lambda\ (x_1)\ e_1)}, env_1), k_1)) \ \to\ (\mathtt{e_1}, env_1[x_1 \mapsto env(x)], k_1)$$

$$(\mathtt{(\lambda\ (x)\ e)}, env, \textbf{fn}((\mathtt{(\lambda\ (x_1)\ e_1)}, env_1), k_1)) \ \to\ (\mathtt{e_1}, env_1[x_1 \mapsto (\mathtt{(\lambda\ (x)\ e)}, env)], k_1)$$

k ::= ... | **let**(x, e, env, k)

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

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

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

# CEK-machine evaluation

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

consider the following question.

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