Lecture 8 — August 26, 2015

- Today:
  - Continuations
  - Continuation Passing Style
- Readings:
  - *TSPL4* (Dybvig) 3.3, 3.4 (http://www.scheme.com/tspl4/)
- Next Week:
  - Lambdas and streams in Java 8
  - Asynchronous JavaScript (Node.js)
  - Declarative programming
  - *SICP* 3.5 [for a treatment of streams in Scheme]
  - *SICP* 4.4 [logical programming]

CSE130, Summer Session II

*Some slides from Matt Might, University of Utah, **matt.might.net**, with permission.*

Using callbacks to return values

- Two values can be returned by returning a list:

```
(define (car&cdr pair)
  (list (car pair) (cdr pair)))
```

```
> (car&cdr '(2 4 6 8))
'(2 (4 6 8))
```

- A ***callback*** is when an earlier joke is referenced
- We can “return” two values by accepting a callback:

```
(define (car&cdr/callback pair receive)
  (receive (car pair) (cdr pair)))
```

```
> (car&cdr/callback '(2 4 6 8) (lambda (x y)
                                 (map (lambda (v) (* v x)) y)))
'(8 12 16)
> (car&cdr/callback '(2 . 3) (lambda (a d)
                               (display (+ d a)) (newline)))
5
> (car&cdr/callback '(2 . 3) list)
'(2 3)
```

Using callbacks to return values

```
(define (car&cdr/callback pair receive)
  (receive (car pair) (cdr pair)))

(define (sum/callback nums cb)
  (cb (apply + nums)))

(define (cons/callback a b cb)
  (cb (cons a b)))
```

```
> (sum/callback (list 3 1 4 1 5 9) display)
23

> (cons/callback #\h '(#\i) list->string)
"hi"

> (car&cdr/callback (cons 2 5) *)
10

> (car&cdr/callback (cons 2 5) +)
7

> (car&cdr/callback (cons 2 5) /)
2/5

> (let ([snoc (lambda (a b) (cons b a))])
    (car&cdrw/callback (cons 2 5) snoc))
'(5 . 2)
```

Control context

"It is the evaluation of actual parameters, not the calling of procedures, that requires creating a control context." —Friedman, et al, *EoPL*, 2001

```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 (* 1 (fact 0)))))
(* 4 (* 3 (* 2 (* 1 1))))
24
```

```
(fact 4)
(ifact 4 1)
(ifact 3 4)
(ifact 2 12)
(ifact 1 24)
(ifact 0 24)
24
```

- Recursive `fact` is invoked in larger and larger ***control contexts:*** This is "the stack" that grows.

Continuations

- Similar to how an environment abstracts data contexts, a ***continuation*** abstracts control contexts.
  - An environment is a collection of names that are associated with locations.
  - A continuation of an expression is: a procedure that takes the result of the expression and completes the computation
- A continuation represents a suspended computation, waiting for a value.

- These phrasings by Matt Might are also helpful:
  - *A continuation is like a saved game.*
  - *They're like time travel.*
  - *The program stack encodes the current continuation.*
  - *The state of a thread is a continuation.*
  - *A continuation is a procedure that never returns to its caller.*
  - *Exceptions are a special case of continuations.*
  - *A continuation is a first-class encoding of control.*

The **`call/cc`** primitive

- The **`call/cc`** procedure is passed another procedure, **`p`**, that has one argument, **`k`**.
- **`call/cc`** obtains the current continuation and applies **`p`** to it.
  - Thus, it doesn't "return" the current continuation, rather
  - **`k`** is the continuation, and we access it from **`p`**.
- Each time **`k`** is applied to a value (if it is ever applied at all):
  - it returns that value to the continuation,
  - becoming the value of the application of the original **`call/cc`**.
- If **`p`** returns without invoking **`k`**, the value returned by **`p`** becomes the value of **`call/cc`**.

Some examples

```
> (call/cc
   (lambda (k)
     (* 5 4)))
20
> (call/cc
   (lambda (k)
     (* 5 (k 4))))
4
> (+ 2
     (call/cc
      (lambda (k)
        (* 5 (k 4)))))
6
```

Non-local exit

- Take the product of a list of numbers: [*TSPL4,* Dybvig]

```
(define (product ls)
  (call/cc
   (lambda (break)
     (let f ([ls ls])
       (cond
         [(null? ls) 1]
         [(= (car ls) 0) (break 0)]
         [else (* (car ls) (f (cdr ls)))])))))
```

```
> (product '(1 2 3 4 5))
120
> (product '(7 3 8 0 19 5))
0
```

Continuations can escape

```
> (let ([x (call/cc (lambda (k) k))])
    x)
#<continuation>
```

```
> (let ([x (call/cc (lambda (k) k))])
    (x "Hello, world"))
application: not a procedure;
 expected a procedure that can be applied to arguments
  given: "Hello, world"
  arguments...:
```

```
> (let ([x (call/cc (lambda (k) k))])
    (x (lambda (dummy)
         "Hello, world")))
"Hello, world"
```

- It's a good job that this isn't confusing in the least

"probably the most confusing Scheme program of its size"

From Dybvig: "it might be easy to guess what it returns, but it takes some thought to figure out why."

```
> (((call/cc (lambda (k) k)) (lambda (x) x)) "HEY!")
"HEY!"
```

Well, we can try to factor it a little:

```
> (define (id x) x)
> (((call/cc id) id) "HEY!")
"HEY!"
```

"The value of the `call/cc` is its own continuation [...]. This is applied to the identity procedure `(lambda (x) x)`, so the `call/cc` returns a second time with this value. Then, the identity procedure is applied to itself, yielding the identity procedure. This is finally applied to `"HEY!"`, yielding `"HEY!"`."

Factorial

- We can capture a continuation and set a global variable to it:

```
(define retry #f)

(define (factorial n)
  (if (= n 0)
      (call/cc (lambda (k) (set! retry k) 1))
      (* n (factorial (- n 1)))))
```

```
> (factorial 4) ⟶ 24

> (retry 1)     ⟶ 24

> (retry 2)     ⟶ 48

> (retry 5)     ⟶ 120
```

- But it only works this way in the evaluator!

```
> (let ()
    (printf "~s " (factorial 4))
    (printf "~s " (retry 5)))
24 120 120 120 120 120 120 120 120 120 ...
```

Note on time travel, Marty

- Calling the continuation does not “turn back time”:
  - Any changes to the heap that have happened in the mean time will remain
  - Any I/O that has been performed of course has already happened

Application: Backtracking

- Find integers x, y, z such that:
  - x, y, z are in the range $[2, 9]$; and
  - $x^2 = y^2 + z^2$

```
> (let ([x (in-range 2 9)]
        [y (in-range 2 9)]
        [z (in-range 2 9)])
    (if (= (* x x)
           (+ (* y y) (* z z)))
        (list x y z)
        (fail)))
'(5 3 4)
```

Application: Backtracking

```
(define (fail)
  (error "no solution"))

(define (in-range a b)
  (call/cc
   (lambda (k)
     (enumerate a b k))))

(define (enumerate a b k)
  (if (> a b)
      (fail)
      (let ([save fail])
        (set! fail
              (lambda ()
                (set! fail save)
                (enumerate (+ a 1) b k)))
        (k a))))
```

Continuation Passing Style

- Like assignment, Scheme doesn't actually *need* **`call/cc`**.
- Instead, we can just make the continuations explicit:

```
(define (product/cb ls return)
  (let ([break return])
    (let f ([ls ls] [return return])
      (cond
        [(null? ls) (return 1)]
        [(= (car ls) 0) (break 0)]
        [else (f (cdr ls)
                 (lambda (x)
                   (return (* (car ls) x))))]))))
```

```
> (product/cb '(1 2 3) (lambda (v) (display v)))
6
> (display (product '(1 2 3)))
6
```

CPS factorial

- We can apply the `-/k` style to our primitives, too
- In CPS, every function call is in the tail position!
- Factorial function, fully converted to CPS style:

```
(define (=/k a b k) (k (= a b)))
(define (-/k a b k) (k (- a b)))
(define (*/k a b k) (k (* a b)))

(define (fact n k) (ifact n 1 k))
(define (ifact n a k)
  (=/k n 0
       (lambda (b)
         (if b
             (k a)
             (-/k n 1
                  (lambda (nm1)
                    (*/k n a (lambda (nta)
                         (ifact nm1 nta k)))))))))
```

```
> (fact 42 display)
1405006117752879898543142606244511569936384000000000
```