Continuations in Scheme

Overview

- Control operators
- Concept of a continuation in a functional programming language
- Scheme's call/cc

Control operators

- Control operators manipulate the order of program steps
- Examples: goto, if , loops, return, break, exit
- A pure functional programming language typically only has one of these: if
  - Well, scheme does have do
- Users can define many of their own control operators with macros (e.g., via define-syntax)
- What about return?

Control in Scheme

- Why doesn't Scheme have a return function?
- Maybe we don't need it to indicate the normal function return spots

```
(define (find-prime1 n)
  ;; returns the first prime ≥ n
  (if (prime? n) n (find-prime1 (add1 n))))
```

- But how about places where we want to *break out* of a computation

```
(define (find-prime2 n)
  ;; returns first prime between n and n**2
  (for-each
      (lambda (x) (and (prime? x) (return x))
      (integers n (* n n))))
```

Catch and Throw in Lisp

- Lisp introduced (in the 70's) *catch* and *throw* to give a non-local return capability
- It was a very useful generalization of return
- (throw <expr>) causes a return from the nearest matching (catch <x>) found on stack

  (defun foo-outer () (catch (foo-inner)))

  (defun foo-inner () … (if x (throw t)) ...)
- Both take an optional tag argument; (throw 'foo) can be caught by (catch 'foo) or (catch)

Scheme's functional approach

- Scheme provides some primitive built-ins that can create these and other control functions
- **call-with-current-continuation** is the main one
  - typically also bound to **call/cc**
- call/cc provides a way to escape out of computation to someplace higher on the stack
- It's used to create other powerful control mechanisms, like *co-routines* and *backtracking*
- call/cc does this in a decidedly functional way

Continuation

- A continuation represents the "future" of a computation at certain moment
- Consider the Scheme expression

  (* (f1 exp1) (f2 **(f3 4)** (f5 exp2)))
- The continuation of **(f3 4)** in that expression is the function

  (lambda (X) (* (f1 exp1) (f2 X (f5 exp2))))
- The **continuation *c*** of an expression ***e*** is a function that awaits the value of ***e*** and proceeds with the computation

Call/cc

- call/cc takes a unary function *f* as its only argument
- The function *f* should be called with the value to be "returned" to the point where call/cc was invoked
- When *f* is called, it reifies the current continuation as an object and applies *f* to it

example

```
> (for-each (lambda (x) ( + x x)) '(1 2 3))
> (for-each (lambda (x) (printf "~s " x)) '(1 2 3))
> 1 2 3 >
> (call/cc
    (lambda (exit)
      (for-each
        (lambda (x) (if (negative? x) (exit x) #f))
        '(54 0 37 -3 245 19)) #t))
-3
```

Implementing return

```
(define (search pred? lst)
 ; returns first item in LST satisfying pred? or #f
 (call/cc
  (lambda (return)
   (for-each
     (lambda (item) (if (pred? item)  (return item)  #f))
       lst)
   #f)))
```

The return can be non-local

```
(define (treat item like-it)
   ; Call like-it with a custom argument when we like item
   (if (good-item? item) (like-it 'fnord) #f))

(define good-item? odd?)

(define (search2 treat lst)
   ; Call treat with every item in lst and a procedure to call
   ; when treat likes this item.
   (call/cc
    (lambda (return) (for-each (lambda (item) (treat item return))
                               lst)
    #f)))
```

We can re-call continuations

```
> (define return #f)
 > (+ 1
      (call/cc (lambda (cont)  (set! return cont) 2))
      3)
6
> return
#<continuation>
> (return 100)
104
```

cont is bound to a continuation (i.e., unary function) that takes a value x and computes (+ 1 x 3)

re-call continuations 2

```
> (define a 1) (define b 2) (define c 3) (define ret #f)
> (define (add-abc)
    (+ a (call/cc (lambda (cont) (set! ret cont) b)) c))
> (add-abc)
6
> ret
#<continuation>
> (ret 100)
104
```

```
> (set! a 1000)
> (ret 100)
104
(set! c 999)
> (ret 100)
1100
```

How do you explain this?

Coroutines

- Coroutines are procedures that persist after they exit and then can be re-entered
- They maintain their state in between calls
- They provide an alternative mechanism to threads for interleaving two processes
- You can implement coroutines in Scheme with continuations

Hefty and Superfluous

```
(define (hefty other)
  (let loop ((n 5))
    (printf "Hefty: ~s\n" n)
    (set! do-other (call/cc other))
    (printf "Hefty (b)\n")
    (set! do-other (call/cc other))
    (printf "Hefty (c)\n")
    (set! do-other (call/cc other))
    (if (> n 0) (loop (- n 1)) #f)))
```

```
(define clock-positions
  '("Straight up." "Quarter after."
    "Half past." "Quarter til."))

(define (superfluous other)
  (let loop ()
    (for-each
      (lambda (graphic)
        (printf "~s\n" graphic)
        (set! other (call/cc other)))
      clock-positions
    (loop)))
```

Hefty and Superfluous

```
> (hefty superfluous)
Hefty: 5
"Straight up."
Hefty (b)
"Quarter after."
Hefty (c)
"Half past."
Hefty: 4
"Quarter til."
...
```

```
...
Hefty (c)
"Half past."
Hefty: 0
"Quarter til."
Hefty (b)
"Straight up."
Hefty (c)
"Quarter after."
#f
```

Summary

- Continuations are a both a bit weird and hard to understand
  - They're also expensive to implement and use
- Most languages choose to add those control features (e.g., return, catch throw) that programmers understand and want
  - These are also added in Scheme via libraries
- But Scheme is mostly a PL for experimenting with PLs and new PL ideas and features