Continuations in Scheme

Scheme provides a built-in mechanism for creating continuations. It has a long name: call-with-current-continuation. This name is usually abbreviated as call/cc (perhaps using define).

call/cc takes a single function as its argument. That function also takes a single argument. That is, we use call/cc as

\[(\text{call/cc } \text{funct})\]

where

\[\text{funct} \equiv (\lambda (\text{con}) (\text{body}))\]

call/cc calls the function that it is given with the “current continuation” as the function’s argument.

Current Continuations

What is the current continuation? It is itself a function of one argument. The current continuation function represents the execution context within which the call/cc appears. The argument to the continuation is a value to be substituted as the return value of call/cc in that execution context.

For example, given

\[(+ \text{(fct n)} 3)\]

the current continuation for \((\text{fct n})\) is

\[\lambda (x) (+ x 3)\]

Given

\[(\ast 2 (+ \text{(fct z)} 10))\]

the current continuation for \((\text{fct z})\) is

\[\lambda (m) (* 2 (+ m 10))\]

To use call/cc to grab a continuation in (say) \((+ \text{(fct n)} 3)\) we make \((\text{fct n})\) the body of a function of one argument. Let’s call that argument return. We therefore create

\[(\lambda (\text{return}) (\text{fct n}))\]

Then

\[(\text{call/cc} \ (\lambda (\text{return}) (\text{fct n})))\]

binds the current continuation to return and executes \((\text{fct n})\).

We can ignore the current continuation bound to return and do a normal return

Or

we can use return to force a return to the calling context of the call/cc.

The call \((\text{return value})\) forces value to be returned as the value of call/cc in its context of call.

Example:

\[(\ast \ (\text{call/cc} \ (\lambda (\text{return}) \ (/ \ (\text{g return}) 0)))) 10\]

\(\text{return}\)

\(\text{define} \ (\text{g con}) \ (\text{con 5})\)

Now during evaluation no divide by zero error occurs. Rather, when \((\text{g return})\) is called, 5 is passed to con, which is bound to return. Therefore 5 is used as the value of the call to call/cc, and 50 is computed.
Continuations are Just Functions

Continuations may be saved in variables or data structures and called in the future to “reactive” a completed or suspended computation.

(define CC ())
(define (F)
  (let ((v (call/cc (lambda (here)
      (set! CC here)
      1)))))
  (display "The ans is: ")
  (display v)
  (newline))

This displays The ans is: 1
At any time in the future, (CC 10) will display The ans is: 10

List Multiplication Revisited

We can use call/cc to reimplement the original *list to force an immediate return of 0 (much like a throw in Java):

(define (*listc L return)
  (cond
    ((null? L) 1)
    ((= 0 (car L)) (return 0))
    (else (* (car L) (*listc (cdr L) return)))))

(define (*list L)
  (call/cc (lambda (return)
    (*listc L return)
  )))

A 0 in L forces a call of (return 0) which makes 0 the value of call/cc.

Interactive Replacement of Error Values

Using continuations, we can also redo *listE so that zeroes can be replaced interactively! Multiple zeroes (in both original and replacement values) are correctly handled.

(define (*liste L return)
  (if (null? L) 1
    (let loop ((value (car L)))
      (if (= 0 value)
        (loop (call/cc (lambda (x) (return x))))
        (* value (*liste (cdr L) return)))))

If a zero is seen, *liste passes back to the caller (via return) a continuation that will set the next value of value. This value is checked, so if it is itself zero, a substitute is requested. Each occurrence of zero forces a return to the caller for a substitute value.