Implementing Coroutines with call/cc

Coroutines are a very handy generalization of subroutines. A coroutine may suspend its execution and later resume from the point of suspension. Unlike subroutines, coroutines do no have to complete their execution before they return.

Coroutines are useful for computation of long or infinite streams of data, where we wish to compute some data, use it, compute additional data, use it, etc.

Subroutines aren’t always able to handle this, as we may need to save a lot of internal state to resume with the correct next value.

Producer/Consumer using Coroutines

The example we will use is one of a consumer of a potentially infinite stream of data. The next integer in the stream (represented as an unbounded list) is read. Call this value n. Then the next n integers are read and summed together. The answer is printed, and the user is asked whether another sum is required. Since we don’t know in advance how many integers will be needed, we’ll use a coroutine to produce the data list in segments, requesting another segment as necessary.

```
(define (consumer)
  (next 0) ; reset next function
  (let loop ((data (moredata)))
    (let ((sum+restoflist
      (sum-n-elems (car data)
        (cons 0 (cdr data)))))
      (display (car sum+restoflist))
      (newline)
      (display "more? ")
      (if (equal? (read) ‘y)
        (if (= 1
          (length sum+restoflist))
          (loop (moredata))
          (loop (cdr sum+restoflist))
        )
        #t ; Normal completion
      )
    ))
)
```

Next, we’ll consider sum-n-elems, which adds the first element of list (a running sum) to the next n elements on the list. We’ll use moredata to extend the data list as needed.

```
(define (sum-n-elems n list)
  (cond
    ((= 0 n) list)
    ((null? (cdr list))
      (sum-n-elems n
        (cons (car list) (moredata))))
    (else
      (sum-n-elems (- n 1)
        (cons (+ (car list)
          (cadr list))
          (cddr list))))
    )
)
```
The function **moredata** is called whenever we need more data. Initially a **producer** function is called to get the initial segment of data. **producer** actually returns the next data segment plus a continuation (stored in **producer-cc**) used to resume execution of **producer** when the next data segment is required.

```
(define moredata
  (let ((producer-cc ()))
    (lambda ()
      (let ((data+cont
                (if (null? producer-cc)
                    (call/cc (lambda (here)
                               (producer here)))
                    (call/cc (lambda (here)
                               (producer-cc here)))))
       (set! producer-cc
dr data+cont)))
      (car data+cont))))
```

Function (**next z**) returns the next z integers in an infinite sequence that starts at 1. A value z=0 is a special flag indicating that the sequence should be reset to start at 1.

```
(define next
  (let ((i 1))
    (lambda (z)
      (if (= 0 z)
          (set! i 1)
          (let loop
                    ((cnt z) (val i) (ints ()
                            (if (> cnt 0)
                                (loop (- cnt 1)
                                      (+ val 1)
                                      (append ints (list val)))
                                (begin
                                      (set! i val)
                                      ints))
                           )
               )
        )))
```

The function **producer** generates an infinite sequence of integers (1,2,3,...). It suspends every 5/10/15/25 elements and returns control to **moredata**.

```
(define (producer initial-return)
  (let loop
    (return initial-return)
    (set! return
      (call/cc (lambda (here)
                 (return (cons (next 5) here))))))
  (set! return
    (call/cc (lambda (here)
               (return (cons (next 10) here))))))
  (set! return
    (call/cc (lambda (here)
               (return (cons (next 15) here))))))
  (loop
    (call/cc (lambda (here)
               (return (cons (next 25) here))))))
  )
```
Lazy Evaluation

Lazy evaluation is sometimes called “call by need.” We do an evaluation when a value is used; not when it is defined.

Scheme provides for lazy evaluation:

(delay expression)

Evaluation of expression is delayed. The call returns a “promise” that is essentially a lambda expression.

(force promise)

A promise, created by a call to delay, is evaluated. If the promise has already been evaluated, the value computed by the first call to force is reused.

Example:

Though and is predefined, writing a correct implementation for it is a bit tricky.

The obvious program

(define (and A B)
  (if A
      B
      #f)
)

is incorrect since B is always evaluated whether it is needed or not. In a call like

(and (not (= i 0)) (> (/ j i) 10))

unnecessary evaluation might be fatal.

An argument to a function is strict if it is always used. Non-strict arguments may cause failure if evaluated unnecessarily.

With lazy evaluation, we can define a more robust and function:

(define (and A B)
  (if A
      (force B)
      #f)
)

This is called as:

(and (not (= i 0))
     (delay (> (/ j i) 10)))

Note that making the programmer remember to add a call to delay is unappealing.
Delayed evaluation also allows us a neat implementation of suspensions. The following definition of an infinite list of integers clearly fails:

\[
\text{(define (inflist } i) \\
\quad \text{(cons } i \text{ (inflist } (+ i 1))))
\]

But with use of delays we get the desired effect in finite time:

\[
\text{(define (inflist } i) \\
\quad \text{(cons } i \text{ (delay (inflist } (+ i 1))))
\]

Now a call like \text{(inflist 1)} creates

\[
1 \quad \text{promise for} \quad \text{(inflist 2)}
\]

We need to slightly modify how we explore suspended infinite lists. We can't redefine \text{car} and \text{cdr} as these are far too fundamental to tamper with.

Instead we'll define \text{head} and \text{tail} to do much the same job:

\[
\text{(define head } \text{car}) \\
\text{(define (tail } L) \\
\quad \text{(force } (\text{cdr } L)))
\]

\text{head} looks at \text{car} values which are fully evaluated.
\text{tail} forces one level of evaluation of a delayed \text{cdr} and saves the evaluated value in place of the suspension (promise).

Given

\[
\text{(define IL (inflist } 1)) \\
\text{(head (tail IL)) returns 2 and expands IL into}
\]

\[
1 \quad \text{promise for} \quad \text{(inflist 3)}
\]