CS342: Organization of Prog. Languages

Topic 12: Continuations

- The rest of the computation as an object
- call/cc
- Using call/cc to implement “return”
- Saving and using continuations
- Output of example
- Using continuations
- Macros for cosmetics
- Example: A mini non-premptive thread system
- Producer-consumer via threads
- Example: Continue and Break

Continuations: The rest of the computation

- Consider the following code fragment:

```
(let ((a #f) (g #f))
  (set! a (list 1 2 3))
  (set! g (+ 3 (car l)))
  (write (list "g is " g)) )
```

- At each point in the evaluation of this expression, some of the expression has been evaluated, and some of it is left to evaluate.
- The part left to evaluate is called the "*current continuation*."
- For example, immediately after `(car l)` is evaluated, the current continuation is to add 3 to the result, set the value of `g` and write some output.

The rest of the computation as an object

- Scheme allows the programmer to capture the unfinished part of the computation as an object.
- This object is called a "*continuation*" and acts like a procedure.
- The primitive to do this is
  `call-with-current-continuation`,
  or `call/cc` for short.

call/cc

- Call/cc takes a function as its argument.
- Call/cc invokes that function passing the current continuation.
- Example.

```
(+ 3 (call/cc  (lambda (r) 4)))
```

The argument function ignores r and returns 4, so the result is 7.

call/cc

- When the current continuation is invoked, it takes a single value as an argument and returns it to the point where the continuation was created.
- Example.

```
(define a <some number>)

(+ 3 (call/cc (lambda (r)
        (if (odd? a) (r 99))
        (write "Yow!") (newline)
        4)))
```

When `r` is invoked, the argument 99 is instantly returned as the result of the `call/cc`.

So the value of the addition is either 7 or 102, depending on whether a is odd or even.

Using call/cc to implement “return”

- We can use call/cc to implement a return statement from anywhere in a function:

```
(define (myfun n) (call/cc (lambda (return)
    (do ((i 1 (+ i 1)))
        ((> i 10) 'Done)

        (if (> i n) (return "Special return"))
        (write "In loop")
        (write  i))

    (write "Done loop")
    "Normal return")))
```

Saving and using continuations

- The real power comes when continuations are saved to resume the compuation at the same point multiple times.

```
(define (myfun)
   (let ((cc (lambda (a) (write "Not yet")))
         (count 0)
         (val #f))

        (write "Hello") (newline)

        (set! val (call/cc
            (lambda (r)
               (set! cc r)
               (write "Hmmmm") (newline)
               888)))

        (write "You again??") (write val)
        (newline)

        (set! count (+ 1 count))
        (if (< count 10) (cc count))

        (write "Goodbye") (newline) ))
```

Output of example

```
"Hello"
"Hmmmm"
"You again??"888
"You again??"1
"You again??"2
"You again??"3
"You again??"4
"You again??"5
"You again??"6
"You again??"7
"You again??"8
"You again??"9
"Goodbye"
```

Using Continuations

- Continuations are very powerful and can be used as the primitive to implement many control structures.
- Think of how they could be used to implement: break, continue, setjmp/longjmp, throw/catch, threads...
- Implementation:
  - Continuations imply that the call/return chain cannot be run as a stack.
  - Control structures implemented by continuations may require compiler optimization to work efficiently.
  - Programs using continuations cannot easily interoperate fully with languages which do not support them.

Using macros – before

```
(let ((L1 #f) (i 1) (prod 1))
    (write "Example 3a")(newline)
 (call/cc (lambda (r) (set! L1 r)))
    (set! prod (* i prod))
    (set! i (+ 1 i))
    (if (<= i 5) (L1 'Dummy))
    (write prod)(newline))
```

Output:

```
"Example 3a"
120
```

Using macros – after

```
(require-library "synrule.ss")

(define-syntax goto
  (syntax-rules ()  ((_ Lab) (Lab 'dummy)) ) )
(define-syntax label
  (syntax-rules () ((_ Lab) (call/cc (lambda (r) (set! Lab r)))) ) )
(define-syntax label-body
  (syntax-rules () ((_ (l1 ...) e1 ...) (let ((l1 #f) ...) e1 ...)) ) )

(label-body (L1)
   (write "Example 3b")(newline)
   (let ((i 1) (prod 1))
     (label L1)
     (set! prod (* i prod))
     (set! i (+ 1 i))
     (if (<= i 5) (goto L1))

     (write prod)(newline)))
```

Output:

```
"Example 3b"
120
```

Example 1: A mini non-premptive thread system

- The set of suspended processes is saved in the list `*task-list*`.

- A new thread is created with the call (`fork` *function*).

  The parameter is a function of zero arguments.

  The fork call does not start the function right away, but simply adds it to the `*task-list*`.

- Threads cooperate by making the call (`sync`) when it is safe or desirable to have a context switch.

- When task is finished, it makes the call (`fini`).

Representation

- A thread shall be represented by a continuation.

  The current point of execution is captured using

```
(let ((this-task (call/cc (lambda (r) r))))
   ...stuff... )
```

- The first time, `this-task` has the continuation `r` as its value.

- When a task is resumed, we pass it the argument `#f`

  Then the above `let` expression is resumed, but `this-task` has the value `#f` this time around and the code `...stuff...` can test for that.

- We use this to control the logic of the thread management.

Some helper functions

```
;; Extend a non-null list with an element at its end.

(define (nconc l val)
   (if (and (pair? l) (eq? (cdr l) '()))
       (set-cdr! l (cons val '()))
       (nconc (cdr l) val)))

(define (writeln . l)
   (write l)
   (newline))
```

The mini threads package

```
(define *task-list* '())

(define (fork fn)
  (let ((this-task (call/cc (lambda (r) r))))
     (if this-task
       (set! *task-list*
             (cons this-task *task-list*))
       (fn) )))

(define (sync)
  (if (pair? *task-list*)
    (let ((this-task (call/cc (lambda (r) r))))
      (if this-task
        (begin
          (nconc *task-list* this-task)
          (let ((next-task (car *task-list*)))
             (set! *task-list* (cdr *task-list*))
             (next-task #f)))) )))

(define (fini)
  (if (pair? *task-list*)
    (let ((next-task (car *task-list*)))
       (set! *task-list* (cdr *task-list*))
       (next-task #f))))
```

Producer-consumer example

```
(define *buf-data* (vector 0 10 20 30 40 50))
(define *buf-used*  6)
(define *buf-avail* 0)

(define (producer)
  (let ((datum 10))
    (do ((niters 1 (+ 1 niters)))
        ((> niters 50) (fini))
      (if (> *buf-avail* 0) (begin
          (set! datum (+ 10 datum))
          (writeln "Producer is inserting " datum)
          (vector-set! *buf-data* *buf-used* datum)
          (set! *buf-avail* (- *buf-avail* 1))
          (set! *buf-used*  (+ *buf-used* 1)) )
        (sync) ) ) ) )
(define (consumer)
  (let ((datum #f))
    (do ((niters 1 (+ 1 niters)))
        ((> niters 50) (fini))
      (if (> *buf-used* 0) (begin
          (set! *buf-avail* (+ *buf-avail* 1))
          (set! *buf-used*  (- *buf-used* 1))
          (set! datum (vector-ref *buf-data* *buf-used*))
          (writeln "Consumer got data " datum) )
        (sync) ) ) ) )
```

Producer-consumer continued

```
(define (doit)
    (fork producer)
    (fork consumer)

    (do ()
        ((not (pair? *task-list*)) 'done)
      (sync) ))
```

Example 2: Loops with Continue and Break

```
(define-syntax my-do
  (syntax-rules ()
    ((_ label init fini expr ...)
     ;; Use a pair to hold two continuations: one for continue, one for break.
     (let ((label (cons 'continue-contn-goes-here 'break-contn-goes-here)))
        (call/cc (lambda (break-contn)
           (set-cdr! label break-contn)
           (do init fini
              (call/cc (lambda (continue-contn)
                 (set-car! label continue-contn)
                 expr ...)) ) )) ) ) ))

(define-syntax my-continue
  (syntax-rules ()
    ((_ label) ((car label) 'nothing)) ) )

(define-syntax my-break
  (syntax-rules ()
    ((_ label value) ((cdr label) value)) ) )
```

```scheme
;; Test:

(define (writeln . l)
   (cond ((pair? l)
           (write (car l)) (apply writeln (cdr l)))
         (else (newline)) ) )

(define (test1)
   (my-do George
       ((i 1 (+ 1 i)))
       ((> i 10) 99)

       (writeln "In loop with i = " i)
       (if (odd? i) (my-continue George))
       (writeln "Still in loop with i = " i) ) )

(define (test2 k)
   (my-do Henry
       ((i 1 (+ 1 i)))
       ((> i 10) 'Normal-Result)

       (if (= i k) (my-break Henry 'Break-Result)) ) )
```

```
> (test1)
"In loop with i = "1
"In loop with i = "2
"Still in loop with i = "2
"In loop with i = "3
"In loop with i = "4
"Still in loop with i = "4
"In loop with i = "5
"In loop with i = "6
"Still in loop with i = "6
"In loop with i = "7
"In loop with i = "8
"Still in loop with i = "8
"In loop with i = "9
"In loop with i = "10
"Still in loop with i = "10
99

> (write (test2  3)) (newline)
break-result

> (write (test2 13)) (newline)
normal-result
```