Today’s Lecture Notes for cs173

sk and dbtucker

September 29, 2000

Quote of the Day: “To compute is human, to continue divine.”—anonymous

Contexts

Consider the following computation in Scheme:

(+ (* 2 3) (- 4 (+ 5 6)))

For each step of computation, let’s look at two parts:

1. the expression currently under evaluation

2. the computation that still needs to be done. This expression depends on
   the result of the current computation, so we’ll represent that hole with a
   ■.

Here is how the expression evaluates:

\[
\begin{align*}
(* 2 3) & \quad (+ □ (- 4 (+ 5 6))) \\
(+ 5 6) & \quad (+ 6 (- 4 □)) \\
(- 4 11) & \quad (+ 6 □) \\
-1 & \quad □
\end{align*}
\]

The evaluations that remain to be done (i.e. the right-hand side above)
are called contexts. We can turn the contexts into legal Scheme expressions by
abstracting the □’s:

\[
\begin{align*}
(* 2 3) & \quad (λ (□) (+ □ (- 4 (+ 5 6)))) \\
(+ 5 6) & \quad (λ (□) (+ 6 (- 4 □))) \\
(- 4 11) & \quad (λ (□) (+ 6 □)) \\
-1 & \quad (λ (□) □)
\end{align*}
\]

Note that when the computation ends, the context is the identity function.
Magic

Now we’ll conjure up a little magic, represented by the Official International Symbol for Magic: \(^\wedge\). Here are some expressions with magical operators. Can you figure out what’s going on?

<table>
<thead>
<tr>
<th>expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>((+ (^* 2 3) 4))</td>
<td>6</td>
</tr>
<tr>
<td>((- 1 (^+ (^* 2 3) 4) 5))</td>
<td>10</td>
</tr>
<tr>
<td>((+ (^* 2 3) (- 4 (^ 5 6))))</td>
<td>6</td>
</tr>
<tr>
<td>((+ (^* 2 3) (^ 4 (^* 5 6))))</td>
<td>30</td>
</tr>
</tbody>
</table>

We’ll call these magical operators (e.g. \(^\wedge\)) escapers—they just evaluate the current expression, but then escape for the remaining computation. In other words, an escaper is a procedure that tosses away its context.

Let the games begin...

Let’s try something easier. We can write a function that computes the product of a list of numbers:

```scheme
(define (prod L)
  (cond
    [(empty? L) 1]
    [(cons? L) (* (first L) (prod (rest L)))])
)
```

No problem there. What happens when we evaluate this expression?

```scheme
(prod '(1 2 3 0 4 5 6))
```

Of course it evaluates to 0, but the Scheme interpreter has to do 7 multiplications to figure that out. We shouldn’t have to do any multiplications if there’s a 0 in the list. Can we modify our code to do so?

```scheme
(define (prod L)
  (cond
    [(empty? L) 1]
    [(cons? L) (cond
      [(zero? (first L)) 0]
      [else (* (first L) (prod (rest L)))]))]
)
```

Now how many multiplications do we perform? When we evaluate the 0, the context is \((\lambda (\_\_\_) (* 1 (\_\_\_ 2 (* 3 \_\_\_))))\), so we still end up doing 3 multiplications.

If only we had one of those escapers, we could avoid those multiplications too. We want something like this:
(define (prod L esc)
  (cond
   [(empty? L) 1]
   [(cons? L) (cond
                [(zero? (first L)) (esc 0)]
                [else (* (first L) (prod (rest L) esc))])])
)

where esc is an escaper.

The Real Thing

We’ll define the function real-prod which creates an escaper and calls prod with it:

(define (real-prod L)
  (let/cc k (prod L k)))

Whoa, what is this let/cc thing? (let/cc k body) turns the current context into an escaper and binds it as k when evaluating body. So let’s see how it works:

(\text{real-prod '}(1 2 3 0 4 5 6))
⇒ (let/cc k (prod ')(1 2 3 0 4 5 6) k), with k = (\lambda \_ \_)
⇒ (((\lambda \_ \_)(* 1 (* 2 (* 3 \_))))(k 0))
⇒ (((\lambda \_ \_)(* 1 (* 2 (* 3 \_))))((\lambda \_ \_)(\_ 0))
⇒ 0

Another example

Let’s rotate lists (to the left):

\text{(rl '}(1 2 3)) = '}(2 3 1)

Can we do it using let/cc?

(define (walk L esc)
  (cond
   [(empty? L) (let/cc last (esc last))]
   [(cons? L) (cons (first L)
                     (walk (rest L) esc))]]
)

(define (rl L)
  (let ([last (let/cc esc
               (walk (rest L) esc))]
         (last (list (first L))))))

This doesn’t quite work.

Exercise: Fix it.
Coroutines

You can also use continuations to define coroutines. Here, A does some computation, then calls B. Then B does some computation and resumes A from the point it called B.

\[
\text{(define (make-coroutine co-body)}
\text{  (letrec ([state (lambda () (co-body resume))]
\text{    [resume (lambda (other)
\text{       (call/cc (lambda (here)
\text{          (set! state here)
\text{          (other))))}))
\text{    (lambda ()
\text{        (state))))))
\text{(define A (make-coroutine
\text{  (lambda (resume)
\text{    (printf "A1\n"
\text{      (resume B)
\text{    (printf "A2\n"
\text{      (resume B)
\text{    (printf "A3\n"
\text{      'done-with-A))))
\text{(define B (make-coroutine
\text{  (lambda (resume)
\text{    (printf "B1\n"
\text{      (resume A)
\text{    (printf "B2\n"
\text{      (resume A)
\text{    (printf "B3\n"
\text{      'done-with-B))))

Funny stuff

The \text{let/cc} form is really just a macro:

\[
\text{(let/cc k body)} \equiv \text{(call/cc (\lambda (k) body))}
\]

Question #1: What is \text{(call/cc call/cc)}?
Question #2: What is \text{((call/cc call/cc) (call/cc call/cc))}?