SICP section 4.2.2

December 25, 2007 at 06:07 Tags SICP

A reminder for folks who've reached this page directly: this is part of mySICP reading and exercise solving project, introduced here. The code for chapter 4 is written in Common Lisp.

Exercise 4.25

In ordinary applicative-order Scheme, with the definition of `unless` as provided, the call `(factorial 5)` will enter an infinite loop. This will happen because this `unless` in applicative-order Scheme evaluates its *exceptional-value* argument in any case, disregarding the value of *condition*. Therefore, no matter what the value of *condition* is, the call `(factorial 5)` executes `(factorial 4)`, which in turn executes `factorial` for `3, 2, 0, -1, -2` ... and so on ad-(minus)-infinitum.

In normal-order evaluation this will work, because `(factorial 1)` won't call `factorial` for 0, as `unless` won't evaluate *exceptional-value* when *condition* is true.

Exercise 4.26

Ben's side:

`unless` can be very easily implemented in our Scheme evaluator, as a derived expression that's transformed into `if`. This information is much simpler than `cond->if` which exists in the evaluator's code. We'll add this clause to `eval.` :

```
  ((unless? exp)
    (eval. (unless->if exp) env))
```

And the implementation is:

```
(defun unless? (exp) (tagged-list? exp 'unless))
(defun unless-predicate (exp) (cadr exp))
(defun unless-consequent (exp)
  (if (not (null (cdddr exp)))
    (cadddr exp)
    'false))
(defun unless-alternative (exp) (caddr exp))

(defun unless->if (exp)
  (make-if
    (unless-predicate exp)
    (unless-consequent exp)
    (unless-alternative exp)))
```

Alyssa's side:

Boy, I can't imagine any really useful application of `unless` as a higher-order procedure. I have no idea what the book authors had in mind here. In Common Lisp, for instance, `unless` is a macro and therefore can not be used as a procedure[1].

Exercise 4.27

```
(define w (id (id 10)))
;;; L-Eval input:
count
;;; L-Eval value:
1
```

Because so far the call to `id` was forced only once, in the outer call. The argument to it `(id 10)` was not evaluated because it’s not needed yet.

```
;;; L-Eval input:
w
;;; L-Eval value:
10
```

The evaluation of `w` forces both calls to `id`. Since the inner call returns 10, the outer does too.

```
;;; L-Eval input:
count
;;; L-Eval value:
2
```

Since, as I mentioned, the evaluation of`w` forced the second `id` call as well, `count` was incremented twice.

Exercise 4.28

Consider this code:

```
(interpret
  '(define adder (lambda (x) (+ x 1))))

(interpret
  '(define (doer func arg)
    (func arg)))

(interpret
  '(doer adder 5))
```

With the way the lazy evaluator is currently implemented, this works and prints 6 as expected. However, had we not forced the operator before passing it to `apply.`, this would throw an error.

Why? Because in execution of`(func arg)` in `doer`, the operator `func` is passed in as an argument. When we call `(doer adder 5)`, the arguments of `doer` are not forced, and so `func` is not an actual value but a thunk.

Exercise 4.29

Even the simple factorial computation function would exhibit worse performance without memoization of arguments:

```
(interpret
  '(define (fact n)
    (if (= n 0)
      1
      (* n (fact (- n 1))))))

(time (interpret '(fact 150))) ; with memoization
=> 0.109 sec

(time (interpret '(fact 150))) ; without memoization
=> 4.5 sec
```

Each time `fact` is called recursively, memoization saves the operation of obtaining the code from the call.

Also, with memoization:

```
(square (id 0))
=> 100
count
=> 1
```

Without memoization:

```
(square (id 0))
=> 100
count
=> 2
```

It's clear why this happens. In `square`, `x` is the same object in both arguments to the multiplication operator. With memoization, its value is computed only once and is obtained from the cache the second time – hence `count` gets incremented only once.

Exercise 4.30

a. `eval.` is called on each expression of the sequence. Since `for-each` uses `begin` which is evaluated as a sequence, each iteration of the `for-each` is evaluated with `eval.` – and since `display` is a primitive procedure, the actual value of its argument is eventually computed.

b. With the original `eval-sequence`:

```
(p1 1)
=> (1 2)
(p2 1)
=> 1
```

Let's see what happens in `p2`. The crucial point here is the call to the internal function `p`. When the call is executed, as with any application of a compound procedure, the body of `p` is evaluated with the environment extended with its arguments tied to their delayed values. The body of `p` is evaluated in sequence. When `e` is evaluated, the assignment thunk is substituted instead of it, but since its value isn't passed to any primitive procedure, it is not forced and the assignment doesn't really happen. So later, when `x` is evaluated and returned, it has its old value. To understand it better, we'll rewrite `p2` as follows[2]:

```
(define (p2 x)
  (define (p e)
    e
    (print e)
    x)
  (p (set! x (cons x '(2)))))
```

Now:

```
(p2 1)
=> OK (1 2)
```

What has happened here ? The call `(print e)` passes `e` to a primitive procedure, and that forces it. While it's being forced, it assigns a new value to `x`.

Using Cy's proposed `eval-sequence`:

```
(p1 1)
=> (1 2)
(p2 1)
=> (1 2)
```

Because `e` is forced in the sequence even without being passed to a primitive procedure.

c. As I explained in the answer to **a.**, the calls to `display` force the arguments anyway because `display` is a primitive procedure. So calling `actual-value` on them can't hurt.

d. I like Cy's approach because it behaves better when there are statements with side effects.

Exercise 4.31

While at first this exercise looks intimidating, it really isn't that difficult once you sit down and start coding. I highly recommend trying to solve it on your own – I found the process very interesting and educational.

First of all, let's add new "syntax recognizers" for the lazy operands of procedure applications. Now each operand can be either an atom (a Lisp name for a "scalar", or non-list: symbols, numbers, etc.) or a list, which means it's a lazy operand:

```
(defun evaluated-operand? (op) (atom op))

(defun lazy-operand? (op)
  (and
    (consp op)
    (or
      (eql (cadr op) 'lazy)
      (eql (cadr op) 'lazy-memo))))

(defun lazy-memo-operand? (op)
  (and (consp op) (eql (cadr op) 'lazy-memo)))

(defun lazy-operand-name (op) (car op))
```

Also, to support both memoized and unmemoized thunks we'll add the `thunk-memo` type:

```
(defun delay-it-memo (exp env) (list 'thunk-memo exp env))
(defun thunk-memo? (obj) (tagged-list? obj 'thunk-memo))
```

And `force-it` will have to be changed to accomodate this difference:

```
(defun force-it (obj)
  (cond ((thunk-memo? obj)
          (let ((result
                  (actual-value
                    (thunk-exp obj)
                    (thunk-env obj))))
            (setf (car obj) 'evaluated-thunk)
            (setf (car (cdr obj)) result)
            (setf (cdr (cdr obj)) '())
            result))
        ((thunk? obj)
          (actual-value (thunk-exp obj) (thunk-env obj)))
        ((evaluated-thunk? obj)
          (thunk-value obj))
        (t obj)))
```

Now, the main change is done only in `apply.`:

```
(defun apply. (proc args env)
  (when (> *evaluator-debug-level* 0)
    (format t "applying ~a to ~a~%" proc args))
  (cond ((primitive-procedure? proc)
          (apply-primitive-procedure
            proc
            (list-of-arg-values args env)))
        ((compound-procedure? proc)
          (eval-sequence
            (procedure-body proc)
            (setup-arguments-env
              (procedure-parameters proc)
              args
              env
              (procedure-env proc))))
        (t
          (error
            "Unknown procedure type in APPLY: " proc))))
```

The difference here is in the call to `eval-sequence` in case of a compound procedure. Instead of directly calling `extend-environment` with a list of delayed arguments, I'm calling a special resolution function that will treat all arguments appropriately. Here it is:

```
(defun setup-arguments-env (operands args args-env proc-env)
  (let ((vars '())
        (vals '()))
    (loop
      for op in operands
      for arg in args
      do
        (cond ((lazy-memo-operand? op)
                (push (lazy-operand-name op) vars)
                (push (delay-it-memo arg args-env) vals))
              ((lazy-operand? op)
                (push (lazy-operand-name op) vars)
                (push (delay-it arg args-env) vals))
              (t ; evaluated-operand?
                (push op vars)
                (push (actual-value arg args-env) vals))))
    (extend-environment vars vals proc-env)))
```

It "pattern matches" the list of operands the of the procedure with the list of actual arguments, and builds up a list of values to extend this environment. For some arguments the values are computed directly, and for some thunks (or memoized thunks) are placed in the list.

Note that I'm not changing `eval.` at all. This is a design decision that could have been different. I decided to resolve the procedure parameter list at the very last moment (when the procedure is applied). This is not the most efficient way to do this (the resolution can be done at definition time) but it's the one requiring the least amount of code. Figuring out such a way for yourself is a great boost to the understanding of the evaluator's code structure – which is exactly why I recommend trying this exercise on your own.

[1] Perhaps this is not a perfect example because CL's `unless` is a bit different from the one proposed in this chapter. Read about it here.

[2] `print` is just like `display` for the sake of our discussion – a primitive procedure.

For comments, please send me an email.

Back to top