CS61A Notes – Week 08a: Analyzing Evaluator

Analyzing Evaluator – "This is where the magic happens"

The intuition is quite easy. We want to "compile" expressions into procedures that take in an environment. This is mainly for speeding up procedure calls (and note, NOT for just recursive procedures).

For instance, in `mc-eval`, let us suppose I use the square procedure a lot. Let's look at the sample call `(square 7)`:

1. `(square 7)`: not self-evaluating, not a symbol...
   application: `eval square`, `eval 7`
   `apply square` to operands: `(7)`
2. `apply`: not primitive, compound procedure
   extend environment, `eval (* x x)`
3. `(* x x)`: not self-evaluating, not a symbol...
   application: `eval *`, `lookup x`, `lookup x`
   `apply *`

Annoying, isn't it? Every time we call `square`, we have to go through `mc-eval`'s cond clause, checking for what type of expression the body of `square` is.

What if we could analyze the `square` procedure once so that we know what type of expression the body of square is? That's what the analyzing evaluator does:

1. `(analyze (square 7)) => <analyzed-procedure>`
   returns analyzed procedure that takes in an environment
2. `(<analyzed-procedure> env) => value`
   applies the body of the procedure to the operands

And that's it! Well, sort of. *How* do we actually do this? First, we analyze the expression. After analysis, we package the information into a procedure:

```
  (lambda (env) (apply (lookup * env)
                       (list (lookup x env) (lookup x env))))
```

Then, every time we call square, we just call the above procedure with the appropriate environment. (**Disclaimer**: This isn't exactly how it looks, because `analyzing-eval` has to handle general cases.)

So here's the model: `(define (analyzing-eval exp env) ((analyze exp) env))`. Analyze the expression, and when it's time for evaluation, plug in the environment.

```
  (define (analyze-lambda exp)
    (let ((vars (cadr exp))
          (analyzed-body (analyze-sequence (lambda-body exp))))
      (lambda (env) (make-procedure vars analyzed-body env))))
```

Here's where most of the benefits of analysis will come. The difference is that we analyze the body BEFORE we make the procedure, so when it comes to calling this procedure, all we have to do is take the analyzed-body and pass in the appropriate environment.

```
  (define (analyze-application exp) ;; for lambda-created procs, not directly
from code
    (lambda (env)
      (let ((analyzed-proc (analyze (operator exp)))
            (analyzed-operands (map (lambda (a) (a env))
                                    (map analyze (operands exp)))))
        ((procedure-body proc)
           (extend-environment (procedure-parameters analyzed-proc)
                               analyzed-operands
                               (procedure-environment analyzed-proc))))
```

So in the example of `square`, it first goes through `analyze-lambda`, then every time we want the value of `square` called with some argument, we use `analyze-application`.

If you haven't noticed by now, all the analyzing evaluator does is **package procedures into regular Scheme procedures**. But the idea is still important – if we can compile procedures into regular Scheme procedures, it's not much harder to compile it into something else, like machine language. This is what compiling is!

(*Note*: When you see `analyze-application` in the code, it won't look like above. But it's just split between `analyze-application` and `execute-application`.)

Which of the following would be faster in `analyzing-evaluator`?

1. **`(+ 1 2)`**

2. **`(((lambda (x) (lambda (y) (+ x y))) 5) 6)`**

3. **`(map (lambda (x) (* x x)) '(1 2 3 4 5 6 7 8 9 10))`**

4. **`(define fib`**
```
     (lambda (n)
       (if (or (= n 0) (= n 1)) 1
           (+ (fib (- n 1)) (fib (- n 2))))))
   (fib 5)
```

5. **`(define fact`**
```
     (lambda (x) (if (= x 0) 1 (* x (fact (- x 1))))))
```

6. **`(accumulate cons nil '(1 2 3 4 5 6 7 8 9 10))`**

FINAL REVIEW QUESTIONS

QUESTION 1.
Write a rule or rules to determine if one integer is less than another. For example, the query

```
     (less ?x (a a a))
```

should give the results

```
     (less () (a a a)) (less (a) (a a a)) (less (a a) (a a a))
```

QUESTION 2.
Consider the following Scheme program:

```
  (let ((a (amb 1 2 3))
        (b (amb 4 5 6)))
    (display "hello")
    (require (= b (* a 2)))
    a)
```

How many times will *hello* be printed? What is the return value?

QUESTION 3. (Question 13 of the Final Exam, Spring 2003)
Given the following definitions:

```
   (define s (make-serializer))
   (define t (make-serializer))
   (define x 10)
   (define (f) (set! x (+ x 3)))
   (define (g) (set! x (* x 2)))
```

Can the following expressions produce an incorrect result, a deadlock, or neither? (By "incorrect result" we mean a result that is not consistent with some sequential ordering of the processes.)

```
   (a) (parallel-execute (s f) (t g))

   (b) (parallel-execute (s f) (s g))

   (c) (parallel-execute (s (t f)) (t g))
   (d) (parallel-execute (s (t f)) (s g))

   (e) (parallel-execute (s (t f)) (t (s g)))
```

QUESTION 4.
Write **subseq** in the query evaluator. **subseq** takes two arguments. The first is a list containing several elements and the second is also a list. The query evaluator should return a solution if the elements in the first list are present in the second list in the same order. For example:

```
>  (subseq (a b c) (a z b y c))  ;; returns a solution
   (subseq (a b c) (a z b y c))
```

QUESTION 5.
Lem E. Tweakit makes the following change to mc-eval in the meta-circular evaluator:

```
     ((application? exp)
      (mc-apply (mc-eval (operator exp) env)
                (list (eval-sequence (operands exp) env)))) ;; <-- changed
        ;; was: (list-of-values (operands exp) env)))
```

He finds that the modified evaluator doesn't work correctly. Indicate which of the following expressions now cause an error, return incorrect results, or return the correct (usual) values. Circle ERROR, INCORRECT, or CORRECT for each.

Assume square is defined as always: (define (square x) (* x x)).

```
ERROR INCORRECT CORRECT (square 1)


ERROR INCORRECT CORRECT (square 2)


ERROR INCORRECT CORRECT (+ 3 4)


ERROR INCORRECT CORRECT (lambda (x y) y)


ERROR INCORRECT CORRECT ((lambda (x y) y) 6 7)


ERROR INCORRECT CORRECT ((lambda (x) (+ x 5)) 9)
```