CS 61A Lecture Notes First Half of Week 7

Topic: Analyzing evaluator, lazy evaluator

Reading: Abelson & Sussman, Sections 4.1.7, 4.2

• **Analyzing Evaluator.** To work with the ideas in this section you should first

```
(load "~cs61a/lib/analyze.scm")
```

in order to get the analyzing metacircular evaluator.

Inefficiency in the Metacircular Evaluator

Suppose we've defined the factorial function as follows:

```
(define (fact num)
  (if (= num 0)
      1
      (* num (fact (- num 1)))))
```

What happens when we compute `(fact 3)`?

```
eval (fact 3)
  self-evaluating?  ==>  #f     if-alternative  ==>  (* num (fact (- num 1)))
  variable?  ==>  #f               eval (* num (fact (- num 1)))
  quoted?  ==>  #f                   self-evaluating?  ==>  #f
  assignment?  ==>  #f               ...
  definition?  ==>  #f               list-of-values (num (fact (- num 1)))
  if?  ==>  #f                         ...
  lambda?  ==>  #f                     eval (fact (- num 1))
  begin?  ==>  #f                        ...
  cond?  ==>  #f                         apply <procedure fact> (2)
  application?  ==>  #t                    eval (if (= num 0) ...)
  eval fact
    self-evaluating?  ==>  #f
    variable?  ==>  #t
    lookup-variable-value  ==>  <procedure fact>
    list-of-values (3)
      eval 3  ==> 3
    apply <procedure fact> (3)
      eval (if (= num 0) ...)
        self-evaluating?  ==>  #f
        variable?  ==>  #f
        quoted?  ==>  #f
        assignment?  ==>  #f
        definition?  ==>  #f
        if?  ==>  #t
          eval-if (if (= num 0) ...)
            if-predicate  ==>  (= num 0)
              eval (= num 0)
                self-evaluating?  ==>  #f
                ...
```

Four separate times, the evaluator has to examine the procedure body, decide that it's an `if` expression, pull out its component parts, and evaluate those parts (which in turn involves deciding what type of expression each part is).

This is one reason why interpreted languages are so much slower than compiled languages: The interpreter does the syntactic analysis of the program over and over again. The compiler does the analysis once, and the compiled program can just do the part of the computation that depends on the actual values of variables.

Separating Analysis from Execution

`Eval` takes two arguments, an expression and an environment. Of those, the expression argument is (obviously!) the same every time we revisit the same expression, whereas the environment will be different each time. For example, when we compute `(fact 3)` we evaluate the body of `fact` in an environment in which `num` has the value 3. That body includes a recursive call to compute `(fact 2)`, in which we evaluate the same body, but now in an environment with `num` bound to 2.

Our plan is to look at the evaluation process, find those parts which depend only on `exp` and not on `env`, and do those only once. The procedure that does this work is called `analyze`.

What is the result of `analyze`? It has to be something that can be combined somehow with an environment in order to return a value. The solution is that `analyze` returns a procedure that takes only `env` as an argument, and does the rest of the evaluation.

Instead of

```
(eval exp env)  ==>  value
```

we now have

```
1.  (analyze exp)  ==>  exp-procedure
2.  (exp-procedure env)  ==>  value
```

When we evaluate the same expression again, we only have to repeat step 2. What we're doing is akin to memoization, in that we remember the result of a computation to avoid having to repeat it. The difference is that now we're remembering something that's only part of the solution to the overall problem, instead of a complete solution.

We can duplicate the effect of the original `eval` this way:

```
(define (eval exp env)
  ((analyze exp) env))
```

The Implementation Details

`Analyze` has a structure similar to that of the original eval:

```
(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ((variable? exp)
         (lookup-var-val exp env))
        ...
        ((foo? exp) (eval-foo exp env))
        ...))
```

```
(define (analyze exp)
  (cond ((self-evaluating? exp)
         (analyze-self-eval exp))
        ((variable? exp)
         (analyze-var exp))
        ...
        ((foo? exp) (analyze-foo exp))
        ...))
```

The difference is that the procedures such as `eval-if` that take an expression and an environment as arguments have been replaced by procedures such as `analyze-if` that take only the expression as argument.

How do these analysis procedures work? As an intermediate step in our understanding, here is a version of `analyze-if` that exactly follows the structure of `eval-if` and doesn't save any time:

```
(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))

(define (analyze-if exp)
  (lambda (env)
    (if (true? (eval (if-predicate exp) env))
        (eval (if-consequent exp) env)
        (eval (if-alternative exp) env))))
```

This version of `analyze-if` returns a procedure with `env` as its argument, whose body is exactly the same as the body of the original `eval-if`. Therefore, if we do

```
((analyze-if some-if-expression) some-environment)
```

the result will be the same as if we'd said

```
(eval-if some-if-expression some-environment)
```

in the original metacircular evaluator.

But we'd like to improve on this first version of `analyze-if` because it doesn't really avoid any work. Each time we call the procedure that `analyze-if` returns, it will do all of the work that the original `eval-if` did.

The first version of `analyze-if` contains three calls to `eval`. Each of those calls does an analysis of an expression and then a computation of the value in the given environment. What we'd like to do is split each of those `eval` calls into its two separate parts, and do the first part only once, not every time:

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env)
      (if (true? (pproc env))
          (cproc env)
          (aproc env)))))
```

In this final version, the procedure returned by `analyze-if` doesn't contain any analysis steps. All of the components were already analyzed before we call that procedure, so no further analysis is needed.

The biggest gain in efficiency comes from the way in which `lambda` expressions are handled. In the original metacircular evaluator, leaving out some of the data abstraction for clarity here, we have

```
(define (eval-lambda exp env)
  (list 'procedure exp env))
```

The evaluator does essentially nothing for a `lambda` expression except to remember the procedure's text and the environment in which it was created. But in the analyzing evaluator we analyze the body of the procedure; what is stored as the representation of the procedure does not include its text! Instead, the evaluator represents a procedure in the metacircular Scheme as a procedure in the underlying Scheme, along with the formal parameters and the defining environment.

Level Confusion

The analyzing evaluator turns an expression such as

```
(if A B C)
```

into a procedure

```
(lambda (env)
  (if (A-execution-procedure env)
      (B-execution-procedure env)
      (C-execution-procedure env)))
```

This may seem like a step backward; we're trying to implement `if` and we end up with a procedure that does an `if`. Isn't this an infinite regress?

No, it isn't. The `if` in the execution procedure is handled by the underlying Scheme, not by the metacircular Scheme. Therefore, there's no regress; we don't call `analyze-if` for that one. Also, the `if` in the underlying Scheme is much faster than having to do the syntactic analysis for the `if` in the meta-Scheme.

So What?

The syntactic analysis of expressions is a large part of what a compiler does. In a sense, this analyzing evaluator is a compiler! It compiles Scheme into Scheme, so it's not a very useful compiler, but it's really not that much harder to compile into something else, such as the machine language of a particular computer.

A compiler whose structure is similar to this one is called a *recursive descent* compiler. Today, in practice, most compilers use a different technique (called a stack machine) because it's possible to automate the writing of a parser that way. (I mentioned this earlier as an example of data-directed programming.) But if you're writing a parser by hand, it's easiest to use recursive descent.

• **Lazy evaluator.** To load the lazy metacircular evaluator, say

```
(load "~cs61a/lib/lazy.scm")
```

Streams require careful attention

To make streams of pairs, the text uses this procedure:

```
;;;;;                          In file cs61a/lectures/4.2/pairs.scm
(define (pairs s t)
  (cons-stream
   (list (stream-car s) (stream-car t))
   (interleave
    (stream-map (lambda (x) (list (stream-car s) x))
                (stream-cdr t))
    (pairs (stream-cdr s) (stream-cdr t)))))
```

In exercise 3.68, Louis Reasoner suggests this simpler version:

```
(define (pairs s t)
  (interleave
   (stream-map (lambda (x) (list (stream-car s) x))
               t)
   (pairs (stream-cdr s) (stream-cdr t))))
```

Of course you know because it's Louis that this doesn't work. But why not? The answer is that `interleave` is an ordinary procedure, so its arguments are evaluated right away, including the recursive call. So there is an infinite recursion before any pairs are generated. The book's version uses `cons-stream`, which is a special form, and so what looks like a recursive call actually isn't—at least not right away.

But in principle Louis is right! His procedure does correctly specify what the desired result should contain. It fails because of a detail in the implementation of streams. In a perfect world, a mathematically correct program such as Louis's version ought to work on the computer.

In section 3.5.4 they solve a similar problem by making the stream programmer use explicit `delay` invocations. (You skipped over that section because it was about calculus.) Here's how Louis could use that technique:

```
(define (interleave-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2)
      (cons-stream
       (stream-car s1)
       (interleave-delayed (force delayed-s2)
                           (delay (stream-cdr s1))))))

(define (pairs s t)
  (interleave-delayed
   (stream-map (lambda (x) (list (stream-car s) x))
               t)
   (delay (pairs (stream-cdr s) (stream-cdr t)))))
```

This works, but it's far too horrible to contemplate; with this technique, the stream programmer has to check carefully every procedure to see what might need to be delayed explicitly. This defeats the object of an abstraction. The user should be able to write a stream program just as if it were a list program, without any idea of how streams are implemented!

Lazy evaluation: delay everything automatically

Back in chapter 1 we learned about *normal order evaluation*, in which argument subexpressions are not evaluated before calling a procedure. In effect, when you type

```
(foo a b c)
```

in a normal order evaluator, it's equivalent to typing

```
(foo (delay a) (delay b) (delay c))
```

in ordinary (applicative order) Scheme. If every argument is automatically delayed, then Louis's `pairs` procedure will work without adding explicit delays.

Louis's program had explicit calls to `force` as well as explicit calls to `delay`. If we're going to make this process automatic, when should we automatically force a promise? The answer is that some primitives need to know the real values of their arguments, e.g., the arithmetic primitives. And of course when Scheme is about to print the value of a top-level expression, we need the real value.

How do we modify the evaluator?

What changes must we make to the metacircular evaluator in order to get normal order?

We've just said that the point at which we want to automatically delay a computation is when an expression is used as an argument to a procedure. Where does the ordinary metacircular evaluator evaluate argument subexpressions? In this excerpt from `eval`:

```
(define (eval exp env)
  (cond ...
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        ...))
```

It's `list-of-values` that recursively calls `eval` for each argument subexpression. Instead we could make thunks:

```
(define (eval exp env)
  (cond ...
        ((application? exp)
         (apply (ACTUAL-VALUE (operator exp) env)
                (LIST-OF-DELAYED-VALUES (operands exp) env)))
        ...))
```

Two things have changed:

1. To find out what procedure to invoke, we use `actual-value` rather than `eval`. In the normal order evaluator, what `eval` returns may be a promise rather than a final value; `actual-value` forces the promise if necessary.

2. Instead of `list-of-values` we call `list-of-delayed-values`. The ordinary version uses `eval` to get the value of each argument expression; the new version will use `delay` to make a list of thunks. (This isn't quite true, and I'll fix it in a few paragraphs.)

When do we want to force the promises? We do it when calling a primitive procedure. That happens in `apply`:

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ...))
```

We change it to force the arguments first:

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure (MAP FORCE ARGUMENTS)))
        ...))
```

Those are the crucial changes. The book gives a few more details: Some special forms must force their arguments, and the read-eval-print loop must force the value it's about to print.

Reinventing `delay` and `force`

I said earlier that I was lying about using `delay` to make thunks. The metacircular evaluator can't use Scheme's built-in `delay` because that would make a thunk in the underlying Scheme environment, and we want a thunk in the metacircular environment. (This is one more example of the idea of level confusion.) Instead, the book uses procedures `delay-it` and `force-it` to implement metacircular thunks.

What's a thunk? It's an expression and an environment in which we should later evaluate it. So we make one by combining an expression with an environment:

```
(define (delay-it exp env)
  (list 'thunk exp env))
```

The rest of the implementation is straightforward.

Notice that the `delay-it` procedure takes an environment as argument; this is because it's part of the implementation of the language, not a user-visible feature. If, instead of a lazy evaluator, we wanted to add a `delay` special form to the ordinary metacircular evaluator, we'd do it by adding this clause to `eval`:

```
((delay? exp) (delay-it (cadr exp) env))
```

Here `exp` represents an expression like `(delay foo)` and so its `cadr` is the thing we really want to delay.

The book's version of `eval` and `apply` in the lazy evaluator is a little different from what I've shown here. My version makes thunks in `eval` and passes them to `apply`. The book's version has `eval` pass the argument expressions to `apply`, without either evaluating or thunking them, and also passes the current environment as a third argument. Then `apply` either evaluates the arguments (for primitives) or thunks them (for non-primitives). Their way is more efficient, but I think this way makes the issues clearer because it's more nearly parallel to the division of labor between `eval` and `apply` in the vanilla metacircular evaluator.

Memoization

Why didn't we choose normal order evaluation for Scheme in the first place? One reason is that it easily leads to redundant computations. When we talked about it in chapter 1, I gave this example:

```
(define (square x) (* x x))

(square (square (+ 2 3)))
```

In a normal order evaluator, this adds 2 to 3 four times!

```
(square (square (+ 2 3)))  ==>
```

```
(* (square (+ 2 3)) (square (+ 2 3)))  ==>
(* (* (+ 2 3) (+ 2 3)) (* (+ 2 3) (+ 2 3)))
```

The solution is memoization. If we force the same thunk more than once, the thunk should remember its value from the first time and not have to repeat the computation. (The four instances of `(+ 2 3)` in the last line above are all the same thunk forced four times, not four separate thunks.)

The details are straightforward; you can read them in the text.