CS 61A Lecture Notes Second Half of Week 6

Topic: Lazy evaluator

Reading: Abelson & Sussman, Sections 4.2

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.