[Go to first, previous, next page; contents; index]

4.2 Variations on a Scheme -- Lazy Evaluation

Now that we have an evaluator expressed as a Lisp program, we can experiment with alternative choices in language design simply by modifying the evaluator. Indeed, new languages are often invented by first writing an evaluator that embeds the new language within an existing high-level language. For example, if we wish to discuss some aspect of a proposed modification to Lisp with another member of the Lisp community, we can supply an evaluator that embodies the change. The recipient can then experiment with the new evaluator and send back comments as further modifications. Not only does the high-level implementation base make it easier to test and debug the evaluator; in addition, the embedding enables the designer to snarf[31] features from the underlying language, just as our embedded Lisp evaluator uses primitives and control structure from the underlying Lisp. Only later (if ever) need the designer go to the trouble of building a complete implementation in a low-level language or in hardware. In this section and the next we explore some variations on Scheme that provide significant additional expressive power.

4.2.1 Normal Order and Applicative Order

In section 1.1, where we began our discussion of models of evaluation, we noted that Scheme is an applicative-order language, namely, that all the arguments to Scheme procedures are evaluated when the procedure is applied. In contrast, normal-order languages delay evaluation of procedure arguments until the actual argument values are needed. Delaying evaluation of procedure arguments until the last possible moment (e.g., until they are required by a primitive operation) is called lazy evaluation.[32] Consider the procedure

```
(define (try a b)
  (if (= a 0) 1 b))
```

Evaluating `(try 0 (/ 1 0))` generates an error in Scheme. With lazy evaluation, there would be no error. Evaluating the expression would return 1, because the argument `(/ 1 0)` would never be evaluated.

An example that exploits lazy evaluation is the definition of a procedure `unless`

```
(define (unless condition usual-value exceptional-value)
  (if condition exceptional-value usual-value))
```

that can be used in expressions such as

```
(unless (= b 0)
        (/ a b)
        (begin (display "exception: returning 0")
               0))
```

This won't work in an applicative-order language because both the usual value and the exceptional value will be evaluated before `unless` is called (compare exercise 1.6). An advantage of lazy evaluation is that some procedures, such as

unless, can do useful computation even if evaluation of some of their arguments would produce errors or would not terminate.

If the body of a procedure is entered before an argument has been evaluated we say that the procedure is non-strict in that argument. If the argument is evaluated before the body of the procedure is entered we say that the procedure is strict in that argument.[33] In a purely applicative-order language, all procedures are strict in each argument. In a purely normal-order language, all compound procedures are non-strict in each argument, and primitive procedures may be either strict or non-strict. There are also languages (see exercise 4.31) that give programmers detailed control over the strictness of the procedures they define.

A striking example of a procedure that can usefully be made non-strict is cons (or, in general, almost any constructor for data structures). One can do useful computation, combining elements to form data structures and operating on the resulting data structures, even if the values of the elements are not known. It makes perfect sense, for instance, to compute the length of a list without knowing the values of the individual elements in the list. We will exploit this idea in section 4.2.3 to implement the streams of chapter 3 as lists formed of non-strict cons pairs.

Exercise 4.25. Suppose that (in ordinary applicative-order Scheme) we define unless as shown above and then define factorial in terms of unless as

```
(define (factorial n)
  (unless (= n 1)
          (* n (factorial (- n 1)))
          1))
```

What happens if we attempt to evaluate (factorial 5)? Will our definitions work in a normal-order language?

Exercise 4.26. Ben Bitdiddle and Alyssa P. Hacker disagree over the importance of lazy evaluation for implementing things such as unless. Ben points out that it's possible to implement unless in applicative order as a special form. Alyssa counters that, if one did that, unless would be merely syntax, not a procedure that could be used in conjunction with higher-order procedures. Fill in the details on both sides of the argument. Show how to implement unless as a derived expression (like cond or let), and give an example of a situation where it might be useful to have unless available as a procedure, rather than as a special form.

4.2.2 An Interpreter with Lazy Evaluation

In this section we will implement a normal-order language that is the same as Scheme except that compound procedures are non-strict in each argument. Primitive procedures will still be strict. It is not difficult to modify the evaluator of section 4.1.1 so that the language it interprets behaves this way. Almost all the required changes center around procedure application.

The basic idea is that, when applying a procedure, the interpreter must determine which arguments are to be evaluated and which are to be delayed. The delayed arguments are not evaluated; instead, they are transformed into objects called

thunks.[34] The thunk must contain the information required to produce the value of the argument when it is needed, as if it had been evaluated at the time of the application. Thus, the thunk must contain the argument expression and the environment in which the procedure application is being evaluated.

The process of evaluating the expression in a thunk is called forcing.[35] In general, a thunk will be forced only when its value is needed: when it is passed to a primitive procedure that will use the value of the thunk; when it is the value of a predicate of a conditional; and when it is the value of an operator that is about to be applied as a procedure. One design choice we have available is whether or not to memoize thunks, as we did with delayed objects in section 3.5.1. With memoization, the first time a thunk is forced, it stores the value that is computed. Subsequent forcings simply return the stored value without repeating the computation. We'll make our interpreter memoize, because this is more efficient for many applications. There are tricky considerations here, however.[36]

Modifying the evaluator

The main difference between the lazy evaluator and the one in section 4.1 is in the handling of procedure applications in `eval` and `apply`.

The `application?` clause of `eval` becomes

```
((application? exp)
 (apply (actual-value (operator exp) env)
        (operands exp)
        env))
```

This is almost the same as the `application?` clause of `eval` in section 4.1.1. For lazy evaluation, however, we call `apply` with the operand expressions, rather than the arguments produced by evaluating them. Since we will need the environment to construct thunks if the arguments are to be delayed, we must pass this as well. We still evaluate the operator, because `apply` needs the actual procedure to be applied in order to dispatch on its type (primitive versus compound) and apply it.

Whenever we need the actual value of an expression, we use

```
(define (actual-value exp env)
  (force-it (eval exp env)))
```

instead of just `eval`, so that if the expression's value is a thunk, it will be forced.

Our new version of `apply` is also almost the same as the version in section 4.1.1. The difference is that `eval` has passed in unevaluated operand expressions: For primitive procedures (which are strict), we evaluate all the arguments before applying the primitive; for compound procedures (which are non-strict) we delay all the arguments before applying the procedure.

```
(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values arguments env)))  ; changed
        ((compound-procedure? procedure)
         (eval-sequence
```

```
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env) ; changed
           (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

The procedures that process the arguments are just like `list-of-values` from section 4.1.1, except that `list-of-delayed-args` delays the arguments instead of evaluating them, and `list-of-arg-values` uses `actual-value` instead of `eval`:

```
(define (list-of-arg-values exps env)
  (if (no-operands? exps)
      '()
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps)
                                env))))
(define (list-of-delayed-args exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps)
                                  env))))
```

The other place we must change the evaluator is in the handling of `if`, where we must use `actual-value` instead of `eval` to get the value of the predicate expression before testing whether it is true or false:

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

Finally, we must change the `driver-loop` procedure (section 4.1.4) to use `actual-value` instead of `eval`, so that if a delayed value is propagated back to the read-eval-print loop, it will be forced before being printed. We also change the prompts to indicate that this is the lazy evaluator:

```
(define input-prompt ";;; L-Eval input:")
(define output-prompt ";;; L-Eval value:")
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output
           (actual-value input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
```

With these changes made, we can start the evaluator and test it. The successful evaluation of the `try` expression discussed in section 4.2.1 indicates that the interpreter is performing lazy evaluation:

```
(define the-global-environment (setup-environment))
(driver-loop)
;;; L-Eval input:
(define (try a b)
  (if (= a 0) 1 b))
;;; L-Eval value:
ok
;;; L-Eval input:
```

```
(try 0 (/ 1 0))
;;; L-Eval value:
1
```

Representing thunks

Our evaluator must arrange to create thunks when procedures are applied to arguments and to force these thunks later. A thunk must package an expression together with the environment, so that the argument can be produced later. To force the thunk, we simply extract the expression and environment from the thunk and evaluate the expression in the environment. We use `actual-value` rather than `eval` so that in case the value of the expression is itself a thunk, we will force that, and so on, until we reach something that is not a thunk:

```
(define (force-it obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj) (thunk-env obj))
      obj))
```

One easy way to package an expression with an environment is to make a list containing the expression and the environment. Thus, we create a thunk as follows:

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

(define (thunk? obj)
  (tagged-list? obj 'thunk))

(define (thunk-exp thunk) (cadr thunk))

(define (thunk-env thunk) (caddr thunk))
```

Actually, what we want for our interpreter is not quite this, but rather thunks that have been memoized. When a thunk is forced, we will turn it into an evaluated thunk by replacing the stored expression with its value and changing the `thunk` tag so that it can be recognized as already evaluated.[37]

```
(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))

(define (thunk-value evaluated-thunk) (cadr evaluated-thunk))
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)  ; replace exp with its value
           (set-cdr! (cdr obj) '())     ; forget unneeded env
           result))
        ((evaluated-thunk? obj)
         (thunk-value obj))
        (else obj)))
```

Notice that the same `delay-it` procedure works both with and without memoization.

Exercise 4.27. Suppose we type in the following definitions to the lazy evaluator:

```
(define count 0)
(define (id x)
  (set! count (+ count 1))
  x)
```

Give the missing values in the following sequence of interactions, and explain your answers.[38]

```
(define w (id (id 10)))
;;; L-Eval input:
count
;;; L-Eval value:
<response>
;;; L-Eval input:
w
;;; L-Eval value:
<response>
;;; L-Eval input:
count
;;; L-Eval value:
<response>
```

Exercise 4.28. `Eval` uses `actual-value` rather than `eval` to evaluate the operator before passing it to `apply`, in order to force the value of the operator. Give an example that demonstrates the need for this forcing.

Exercise 4.29. Exhibit a program that you would expect to run much more slowly without memoization than with memoization. Also, consider the following interaction, where the `id` procedure is defined as in exercise 4.27 and `count` starts at 0:

```
(define (square x)
  (* x x))
;;; L-Eval input:
(square (id 10))
;;; L-Eval value:
<response>
;;; L-Eval input:
count
;;; L-Eval value:
<response>
```

Give the responses both when the evaluator memoizes and when it does not.

Exercise 4.30. Cy D. Fect, a reformed C programmer, is worried that some side effects may never take place, because the lazy evaluator doesn't force the expressions in a sequence. Since the value of an expression in a sequence other than the last one is not used (the expression is there only for its effect, such as assigning to a variable or printing), there can be no subsequent use of this value (e.g., as an argument to a primitive procedure) that will cause it to be forced. Cy thus thinks that when evaluating sequences, we must force all expressions in the sequence except the final one. He proposes to modify `eval-sequence` from section 4.1.1 to use `actual-value` rather than `eval`:

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (actual-value (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

a. Ben Bitdiddle thinks Cy is wrong. He shows Cy the `for-each` procedure described in exercise 2.23, which gives an important example of a sequence with side effects:

```
(define (for-each proc items)
  (if (null? items)
      'done
      (begin (proc (car items))
             (for-each proc (cdr items)))))
```

He claims that the evaluator in the text (with the original `eval-sequence`) handles this correctly:

```
;;; L-Eval input:
(for-each (lambda (x) (newline) (display x))
          (list 57 321 88))
57
321
88
;;; L-Eval value:
done
```

Explain why Ben is right about the behavior of `for-each`.

b. Cy agrees that Ben is right about the `for-each` example, but says that that's not the kind of program he was thinking about when he proposed his change to `eval-sequence`. He defines the following two procedures in the lazy evaluator:

```
(define (p1 x)
  (set! x (cons x '(2)))
  x)

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

What are the values of `(p1 1)` and `(p2 1)` with the original `eval-sequence`? What would the values be with Cy's proposed change to `eval-sequence`?

c. Cy also points out that changing `eval-sequence` as he proposes does not affect the behavior of the example in part a. Explain why this is true.

d. How do you think sequences ought to be treated in the lazy evaluator? Do you like Cy's approach, the approach in the text, or some other approach?

Exercise 4.31. The approach taken in this section is somewhat unpleasant, because it makes an incompatible change to Scheme. It might be nicer to implement lazy evaluation as an upward-compatible extension, that is, so that ordinary Scheme programs will work as before. We can do this by extending the syntax of procedure declarations to let the user control whether or not arguments are to be delayed. While we're at it, we may as well also give the user the choice between delaying with and without memoization. For example, the definition

```
(define (f a (b lazy) c (d lazy-memo))
  ...)
```

would define `f` to be a procedure of four arguments, where the first and third arguments are evaluated when the procedure is called, the second argument is delayed, and the fourth argument is both delayed and memoized. Thus, ordinary procedure definitions will produce the same behavior as ordinary Scheme, while adding the `lazy-memo` declaration to each parameter of every compound procedure will produce the behavior of the lazy evaluator defined in this section. Design and implement the changes required to produce such an extension to Scheme. You will have to implement new syntax procedures to handle the new syntax for `define`. You must also arrange for `eval` or `apply` to determine when arguments are to be delayed, and to force or delay arguments accordingly, and you must arrange for forcing to memoize or not, as appropriate.

4.2.3 Streams as Lazy Lists

In section 3.5.1, we showed how to implement streams as delayed lists. We introduced special forms `delay` and `cons-stream`, which allowed us to construct a ``promise'' to compute the `cdr` of a stream, without actually fulfilling that promise until later. We could use this general technique of introducing special forms whenever we need more control over the evaluation process, but this is awkward. For one thing, a special form is not a first-class object like a procedure, so we cannot use it together with higher-order procedures.[39] Additionally, we were forced to create streams as a new kind of data object similar but not identical to lists, and this required us to reimplement many ordinary list operations (`map`, `append`, and so on) for use with streams.

With lazy evaluation, streams and lists can be identical, so there is no need for special forms or for separate list and stream operations. All we need to do is to arrange matters so that `cons` is non-strict. One way to accomplish this is to extend the lazy evaluator to allow for non-strict primitives, and to implement `cons` as one of these. An easier way is to recall (section 2.1.3) that there is no fundamental need to implement `cons` as a primitive at all. Instead, we can represent pairs as procedures:[40]

```
(define (cons x y)
  (lambda (m) (m x y)))
(define (car z)
  (z (lambda (p q) p)))
(define (cdr z)
  (z (lambda (p q) q)))
```

In terms of these basic operations, the standard definitions of the list operations will work with infinite lists (streams) as well as finite ones, and the stream operations can be implemented as list operations. Here are some examples:

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
(define (map proc items)
  (if (null? items)
      '()
      (cons (proc (car items))
            (map proc (cdr items)))))
(define (scale-list items factor)
  (map (lambda (x) (* x factor))
```

```
      items))
(define (add-lists list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        (else (cons (+ (car list1) (car list2))
                    (add-lists (cdr list1) (cdr list2))))))
(define ones (cons 1 ones))
(define integers (cons 1 (add-lists ones integers)))
;;; L-Eval input:
(list-ref integers 17)
;;; L-Eval value:
18
```

Note that these lazy lists are even lazier than the streams of chapter 3: The `car` of the list, as well as the `cdr`, is delayed.[41] In fact, even accessing the `car` or `cdr` of a lazy pair need not force the value of a list element. The value will be forced only when it is really needed -- e.g., for use as the argument of a primitive, or to be printed as an answer.

Lazy pairs also help with the problem that arose with streams in section 3.5.4, where we found that formulating stream models of systems with loops may require us to sprinkle our programs with explicit `delay` operations, beyond the ones supplied by `cons-stream`. With lazy evaluation, all arguments to procedures are delayed uniformly. For instance, we can implement procedures to integrate lists and solve differential equations as we originally intended in section 3.5.4:

```
(define (integral integrand initial-value dt)
  (define int
    (cons initial-value
          (add-lists (scale-list integrand dt)
                     int)))
  int)
(define (solve f y0 dt)
  (define y (integral dy y0 dt))
  (define dy (map f y))
  y)
;;; L-Eval input:
(list-ref (solve (lambda (x) x) 1 0.001) 1000)
;;; L-Eval value:
2.716924
```

Exercise 4.32. Give some examples that illustrate the difference between the streams of chapter 3 and the ``lazier'' lazy lists described in this section. How can you take advantage of this extra laziness?

Exercise 4.33. Ben Bitdiddle tests the lazy list implementation given above by evaluating the expression

```
(car '(a b c))
```

To his surprise, this produces an error. After some thought, he realizes that the ``lists'' obtained by reading in quoted expressions are different from the lists manipulated by the new definitions of `cons`, `car`, and `cdr`. Modify the evaluator's treatment of quoted expressions so that quoted lists typed at the driver loop will produce true lazy lists.

Exercise 4.34. Modify the driver loop for the evaluator so that lazy pairs and lists will print in some reasonable way. (What are you going to do about infinite lists?)

You may also need to modify the representation of lazy pairs so that the evaluator can identify them in order to print them.

---

[31] Snarf: ``To grab, especially a large document or file for the purpose of using it either with or without the owner's permission.'' Snarf down: ``To snarf, sometimes with the connotation of absorbing, processing, or understanding.'' (These definitions were snarfed from Steele et al. 1983. See also Raymond 1993.)

[32] The difference between the ``lazy'' terminology and the ``normal-order'' terminology is somewhat fuzzy. Generally, ``lazy'' refers to the mechanisms of particular evaluators, while ``normal-order'' refers to the semantics of languages, independent of any particular evaluation strategy. But this is not a hard-and-fast distinction, and the two terminologies are often used interchangeably.

[33] The ``strict'' versus ``non-strict'' terminology means essentially the same thing as ``applicative-order'' versus ``normal-order,'' except that it refers to individual procedures and arguments rather than to the language as a whole. At a conference on programming languages you might hear someone say, ``The normal-order language Hassle has certain strict primitives. Other procedures take their arguments by lazy evaluation.''

[34] The word thunk was invented by an informal working group that was discussing the implementation of call-by-name in Algol 60. They observed that most of the analysis of (``thinking about'') the expression could be done at compile time; thus, at run time, the expression would already have been ``thunk'' about (Ingerman et al. 1960).

[35] This is analogous to the use of `force` on the delayed objects that were introduced in chapter 3 to represent streams. The critical difference between what we are doing here and what we did in chapter 3 is that we are building delaying and forcing into the evaluator, and thus making this uniform and automatic throughout the language.

[36] Lazy evaluation combined with memoization is sometimes referred to as call-by-need argument passing, in contrast to call-by-name argument passing. (Call-by-name, introduced in Algol 60, is similar to non-memoized lazy evaluation.) As language designers, we can build our evaluator to memoize, not to memoize, or leave this an option for programmers (exercise 4.31). As you might expect from chapter 3, these choices raise issues that become both subtle and confusing in the presence of assignments. (See exercises 4.27 and 4.29.) An excellent article by Clinger (1982) attempts to clarify the multiple dimensions of confusion that arise here.

[37] Notice that we also erase the `env` from the thunk once the expression's value has been computed. This makes no difference in the values returned by the interpreter. It does help save space, however, because removing the reference from the thunk to the `env` once it is no longer needed allows this structure to be garbage-collected and its space recycled, as we will discuss in section 5.3.

Similarly, we could have allowed unneeded environments in the memoized delayed objects of section 3.5.1 to be garbage-collected, by having `memo-proc` do something like `(set! proc '())` to discard the procedure `proc` (which includes the environment in which the `delay` was evaluated) after storing its value.

[38] This exercise demonstrates that the interaction between lazy evaluation and side effects can be very confusing. This is just what you might expect from the discussion in chapter 3.

[39] This is precisely the issue with the `unless` procedure, as in exercise 4.26.

[40] This is the procedural representation described in exercise 2.4. Essentially any procedural representation (e.g., a message-passing implementation) would do as well. Notice that we can install these definitions in the lazy evaluator simply by typing them at the driver loop. If we had originally included `cons`, `car`, and `cdr` as primitives in the global environment, they will be redefined. (Also see exercises 4.33 and 4.34.)

[41] This permits us to create delayed versions of more general kinds of list structures, not just sequences. Hughes 1990 discusses some applications of ``lazy trees.''

[Go to first, previous, next page; contents; index]