Chapter 13

Lazy Evaluation

Ordinary men and women, having the opportunity of a happy life, will become more kindly and less persecuting and less inclined to view others with suspicion. The taste for war will die out, partly for this reason, and partly because it will involve long and severe work for all. Good nature is, of all moral qualities, the one that the world needs most, and good nature is the result of ease and security, not of a life of arduous struggle. Modern methods of production have given us the possibility of ease and security for all; we have chosen, instead, to have overwork for some and starvation for others. Hitherto we have continued to be as energetic as we were before there were machines; in this we have been foolish, but there is no reason to go on being foolish forever.

Bertrand Russell (co-author of *Principia Mathematica*), *In Praise of Idleness*, 1932

By changing the language interpreter, we can extend the programming language and change the evaluation rules. This can enable constructs that could not be expressed in the previous language, and change the way we think about solving problems. In this chapter, we explore a variation to our Charme interpreter in which the evaluation rule for application is altered so the expressions passed as parameters are not evaluated until their values are needed. This is known as *lazy evaluation*, and it enables procedures to be defined that could not be defined using the normal (eager) evaluation rule.

13.1 In Praise of Laziness

The original Charme interpreter, and the standard Scheme language, evaluates procedure arguments eagerly: all argument subexpressions are evaluated whether or not their values are needed. This is why, for example, we need a special form for if-expressions rather than being able to define a procedure `if` with the same behavior. With the normal Scheme (and Charme) evaluation rules, the following procedure would not have the same behavior as the if-expression special form:

```
(define if
   (lambda (p c a)
      (cond (p c)
            (#t a))))
```

For uses where the consequent and alternate expressions can be evaluated without producing an error, having a side-effect, or failing to terminate, the `if` procedure behaves indistinguishably from the special form. For example,

```
> (if (> 3 4) 12 13)
13
```

If it is possible to tell if one of the expressions is evaluated, however, the `if` procedure behaves very differently from the if-expression special form:

```
> (if (> 3 4) (car null) 13)
♯ car: expects argument of type <pair>; given ()
```

With the special form if-expression, the consequent expression is only evaluated if the predicate expression is true. Hence, the expression above would evaluate to `13` with the special form if expression provided by Scheme.

By changing the Charme evaluator to delay evaluation of operand expressions until their value is needed, we can enable programs to define procedures that conditionally evaluate their arguments. This is known as *lazy evaluation*, since an expression is not evaluated until its value is needed. Confusingly, it is also known as *normal order* evaluation, even though in the Scheme language it is not the normal evaluation order. Scheme is an *applicative-order* language, which means that

all arguments are evaluated as part of the application rule, whether or not their values are needed by the called procedure. Other languages including Haskell and Miranda provide lazy evaluation as the standard application rule.

Lazy evaluation has several advantages over eager evaluation. As the `if` example indicates, it is possible to express procedures in a language with lazy evaluation that cannot be expressed in a language that has eager evaluation. Many of the special forms in Scheme including `if`, `cond`, and `begin` could be defined as regular procedures in a language with lazy evaluation. It may also allow some evaluations to be performed more efficiently. As an extreme example, consider the expression, `((lambda (x) 3) (loop-forever))`, where `loop-forever` is defined as:

```
(define loop-forever (lambda () (loop-forever)))
```

With lazy evaluation the application expression evaluates to 3; with eager evaluation, the evaluation never terminates since the procedure is not applied until after its operand expressions finish evaluating, but the `(loop-forever)` expression never finishes evaluating.

We will encourage you to develop the three great virtues of a programmer: Laziness, Impatience, and Hubris.
Larry Wall, *Programming Perl*

13.2 Delaying Evaluation

To implement lazy evaluation in our interpreter we need to modify the application expression evaluation rule to delay evaluating the operand expressions until they are needed. To do this, we introduce a new datatype known as a *thunk*. We define a Python class, `Thunk` for representing thunks. A thunk keeps track of an expression whose evaluation is delayed until it is needed. We want to ensure that once the evaluation is performed, the resulting value is saved so the expression does not need to be evaluated again. Thus, a `Thunk` is in one of two possible states: *unevaluated* (the operand expression has not yet been needed, so it has not been evaluated and its value is unknown), and *evaluated* (the operand expression's value has been needed at least once, and its known value is recorded). In addition to changing the application evaluation rule to record the operand expressions as `Thunk` objects, we need to alter the rest of the evaluation rules to deal with `Thunk` objects.

The `Thunk` class implements thunks. To delay evaluation of an expression, it keeps track of the expression. Since the value of the expression may be needed when the evaluator is evaluating an expression in some other environment, we also need to keep track of the environment in which the thunk expression should be evaluated.

```
class Thunk:
    def __init__(self, expr, env):
        self._expr = expr
        self._env = env
        self._evaluated = False
    def value(self):
        if not self._evaluated:
            self._value = forceeval(self._expr, self._env)
            self._evaluated = True
        return self._value

def isThunk(expr):
    return isinstance(expr, Thunk)
```

The implementation uses the `_evaluated` instance variable, to keep track of whether or not the thunk expression has been evaluated. Initially this value is `False`. The `_value` instance variable keeps track of the value of the thunk once it has been evaluated.

To implement lazy evaluation, we change the evaluator so there are two different evaluation procedures: `meval` is the standard evaluation procedure (which does not evaluate thunks), and `forceeval` is the evaluation procedure that forces thunks to be evaluated to values. This means the interpreter should use `meval` when the actual expression value may not be needed, and `forceeval` only when the value of the expression is needed. We need to force evaluation when the result is displayed to the user in the `evalLoop` procedure, hence, the call to `meval` in the `evalLoop` procedure is replaced with:

```
    res = forceeval(expr, globalEnvironment)
```

The `meval` procedure is modified to add an `elif` clause for thunk objects that

returns the same expression:

```
def meval(expr, env):
    ... # same as before
    elif isThunk(expr): # Added to support
       return expr # lazy evaluation
    else:
       evalError ("Unknown expression type: " + str(expr))
```

The `forceeval` procedure first uses `meval` to evaluate the expression normally. If the result is a thunk, it uses the `value` method to force evaluation of the thunk expression. Recall that the `Thunk.value` method itself uses `forceeval` to find the result of the thunk expression, so there is no need to recursively evaluated the value resulting from the `value` invocation.

```
def forceeval(expr, env):
    value = meval(expr, env)
    if isinstance(value, Thunk):
        return value.value()
    else:
        return value
```

To change the rule for evaluating application expressions to support delayed evaluation of operands, we need to redefine the `evalApplication` procedure. Instead of evaluating all the operand subexpressions, the new procedure creates `Thunk` objects representing each operand. Only the first subexpression, that is, the procedure to be applied, must be evaluated. Note that `evalApplication` uses `forceeval` to obtain the value of the first subexpression, since its actual value is needed in order to apply it.

```
def evalApplication(expr, env):
    ops = map (lambda sexpr: Thunk(sexpr, env), expr[1:])
    return mapply(forceeval(expr[0], env), ops)
```

There are two other places where actual values are needed: when applying a primitive procedure and when making a decision that depends on a program value. The first situation requires actual values since the primitive procedures are not defined to operate on thunks. To apply a primitive, we need the actual values of its operands, so must force evaluation of any thunks in the operands. Hence, the definition for `mapply` forces evaluation of the operands to a primitive procedure using the `deThunk` procedure.

```
def deThunk(expr):
    if isThunk(expr):
        return expr.value()
    else:
        return expr

def mapply(proc, operands):
    if (isPrimitiveProcedure(proc)):
        ops = map (lambda op: deThunk(op), ops)
        return proc(ops)
    elif ... # same as before
```

The second situation arises in the evaluation rule for conditional expressions. In order to know how to evaluate the conditional, it is necessary to know the actual value of the predicate expressions. Without knowing if the predicate evaluates to a true or false value, the evaluator cannot proceed correctly. It must either evaluate the consequent expression associated with the predicate, or continue to evaluate the following predicate expression. So, we change the `evalConditional` procedure to use:

```
result = forceeval(predicate, env)
```

This forces the predicate to evaluate to a value (even if it is a thunk), so its actual value can be used to determine how the rest of the conditional expression evaluates.

13.3 Lazy Programming

Lazy evaluation enables programming constructs that are not possible with eager evaluation. For example, the `if` procedure defined at the beginning of this chapter behaves like the if-expression special form in Scheme only if our interpreter has lazy evaluation. Lazy evaluation also enables programs to deal with seemingly infinite data structures. This is possible since only those values of the apparently infinite data structure that are used need to be created.

Much of my work has come from being lazy.
John Backus

Suppose we define LazyCharme procedures similar to the Scheme procedures for manipulating pairs:

```
(define cons
   (lambda (a b)
       (lambda (p) (if p a b))))

(define car (lambda (p) (p #t)))
(define cdr (lambda (p) (p #f)))
```

These behave similarly to the corresponding Scheme procedures, except their operands are evaluated lazily. This means, we can define an infinite list:

```
(define ints-from
    (lambda (n)
       (cons n (ints-from (+ n 1)))))
```

In Scheme (with eager evaluation), `(ints-from 1)` would never finish evaluating. It constructs a list of all integers starting at 1, but has no base case for stopping the recursive applications. In LazyCharme, however, the operands to the `cons` application in the body of `ints-from` are not evaluated until they are needed. Hence, `(ints-from 1)` terminates. It produces a seemingly infinite list, but only the evaluations that are needed are performed:

```
LazyCharme> (car (ints-from 1))
1
```

```
LazyCharme> (car (cdr (cdr (cdr (ints-from 1)))))
4
```

Some evaluations will still fail to terminate. For example, using the standard definition of `length`:

```
(define null #f)
(define null?
   (lambda (x) (= x #f)))

(define length
   (lambda (lst)
      (if (null? lst) 0
          (+ 1 (length (cdr lst))))))
```

Evaluating `(length (ints-from 1))` would never terminate. Every time we evaluate an application of `length`, it applies `cdr` to the input list, which causes `ints-from` to evaluate another `cons`. The actual length of the list is infinite, so the application of `length` does not terminate.

We can use lists with delayed evaluation to solve problems. Reconsider the Fibonacci sequence from Chapter 6. Using lazy evaluation, we can define a list that is the infinitely long Fibonacci sequence:[1]

```
(define fibo-gen
   (lambda (a b)
      (cons a (fibo-gen b (+ a b)))))

(define fibos (fibo-gen 0 1))
```

[1] This example is based on *Structure and Interpretation of Computer Programs*, Section 3.5.2, which also presents several other examples of interesting programs constructed using delayed evaluation.

Then, to obtain the n_{th} Fibonacci number, we just need to get the n_{th} element of `fibos`:

```
(define fibo
   (lambda (n) (get-nth fibos n)))
```

where `get-nth` is defined as:

```
(define get-nth
    (lambda (lst n)
        (if (= n 0)
            (car lst)
            (get-nth (cdr lst) (- n 1)))))
```

Another strategy for defining the Fibonacci sequence is to first define a procedure that merges two (possibly infinite) lists, and then define the Fibonacci sequence in terms of itself. The `merge-lists` procedure combines elements in two lists using an input procedure.

```
(define merge-lists
   (lambda (lst1 lst2 proc)
      (if (null? lst1) null
          (if (null? lst2) null
              (cons (proc (car lst1) (car lst2))
                    (merge-lists (cdr lst1) (cdr lst2) proc))))))
```

We can think of the Fibonacci sequence as the combination of two sequences, starting with the `0` and `1` base cases, combined using addition where the second sequences is offset by one position. This allows us to define the Fibonacci sequence without needing a separate generator procedure:

```
(define fibos
   (cons 0 (cons 1 (merge-lists fibos (cdr fibos) +))))
```

The sequence is defined to start with 0 and 1 as the first two elements. The following elements are the result of merging `fibos` and `(cdr fibos)` using the + procedure. So, the third element in the sequence is `(+ (car fibos) (car (cdr fibos)))` which evaluates to 1, and the fourth element is `(+ (car (cdr fibos)) (car (cdr (cdr fibos))))` which evaluates to 2. This definition relies heavily on lazy evaluation; otherwise, the evaluation of

```
(merge-lists fibos (cdr fibos) +)
```

would never terminate: the input lists are effectively infinite.

Exercise 13.1. Define the sequence of factorials using techniques similar to how we defined `fibos`. ♢

Exercise 13.2. For each of these questions, try to figure out what infinite list is defined by the given expression without evaluating it in LazyCharme.

a. `(define p (cons 1 (merge-lists p p +)))`

b. `(define t (cons 1 (merge-lists t (merge-lists t t +) +)))`

c. `(define twos (cons 2 twos))`

d. `(merge-lists (ints-from 1) twos *)`

♢

Exercise 13.3. Assuming the definitions from the previous exercise, what is the value of `dl`?

```
(define filter
   (lambda (lst proc)
      (if (null? lst) null
          (if (proc (car lst))
```

```
              (cons (car lst) (filter (cdr lst) proc))
              (filter (cdr lst) proc)))))

(define contains-ordered
   (lambda (lst val)
      (if (null? lst) #f
          (if (> (car lst) val) #f
              (if (= (car lst) val) #t
                  (contains-ordered (cdr lst) val))))))

(define dl
   (filter (ints-from 1)
           (lambda (el)
              (if (contains-ordered
                     (merge-lists (ints-from 1) twos *)
                     el)
                  #f
                  #t))))
```

♢

Exercise 13.4.(⋆⋆) A simple algorithm known as the "Sieve of Eratosthenes" for finding prime numbers was created by Eratosthenes, an ancient Greek mathematician and astronomer (he was also known for calculating the circumference of the Earth). The algorithm imagines starting with an (infinite) list of all the integers starting from 2. Then, it imagines repeating the following two steps forever:

1. Circle the first number in the list that is not crossed off. That number is prime.
2. Cross off all numbers in the list that are multiples of the circled number.

To carry out the algorithm in practice, of course, the initial list of numbers must be finite, otherwise it would take forever to cross off all the multiples of 2.

Implement the sieve algorithm using lists with lazy evaluation. You may find the `filter` and `merge-lists` procedures useful, but will probably find it necessary to define some additional procedures. ♢

13.4 Summary

We can produce a new language by changing the evaluation rules of an interpreter. Changing the evaluation rules changes what programs mean, and may enable programmers to approach problems in new ways. In this example, we have seen that changing the evaluation rule for applications to evaluate operand expressions lazily enables new programming constructs.