13

Laziness

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, *In Praise of Idleness*, 1932

By changing the language interpreter, we can change the evaluation rules of the programming language. This enables a new problem-solving strategy: if the solution to a problem cannot be expressed easily in an existing language, define and implement an interpreter for a new language in which the problem can be solved more easily.

In this chapter, we explore a variation on Charme we call *LazyCharme*. LazyCharme changes the application evaluation rule so that operand expressions are not evaluated until their values are needed. This is known as *lazy evaluation*.

lazy evaluation

Lazy evaluation enables many procedures which would otherwise be awkward to express to be defined concisely. Since both Charme and LazyCharme are universal programming languages they can express the same set of computations: all of the procedures we define that take advantage of lazy evaluation could be defined with eager evaluation (for example, by first defining an interpreter for a lazy evaluation language as we do in this chapter).

13.1 Lazy Evaluation

The Charme interpreter defined in the previous chapter as well as the standard Scheme language evaluate application expressions *eagerly*: all operand subexpressions are evaluated whether or not their values are needed. For example, evaluating ((**lambda** (*a*) 3) (*loop-forever*)) (where *loop-forever* is a procedure that never terminates) will not finish.

Eager evaluation means that any expression which does not evaluate all its

sub-expressions must be a special form. For example, there is no way to define a procedure that behaves like the if special form. We could attempt to define such a procedure by first defining *true* and *false* as procedures that take two parameters and output the first or second parameter:

(**define** *true* (**lambda** (*a b*) *a*))
(**define** *false* (**lambda** (*a b*) *b*))

These definitions provide the expected behavior similar to the **if** special form with this definition of *ifp*:

(**define** *ifp* (**lambda** (*p c a*) (*p c a*)))

For example, (*ifp true* 3 4) evaluates to 3 and (*ifp false* 3 4) evaluates to 4.

For uses where evaluating the consequent and alternate expressions does not produce an error, side-effect, or take a noticeable amount of time, the *ifp* procedure behaves indistinguishably from the special form (this assumes the values for *true* and *false* produced by the primitive procedures are changed accordingly to produce the values of *true* and *false* defined above).

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

When it is possible to tell if the alternate or consequent expressions is evaluated, however, the *ifp* procedure behaves very differently from the if-expression special form. For example, (*ifp false* (*cdr null*) 1) evaluates to an error because (*cdr null*) attempts to apply *cdr* to a value that is not a Pair. With the if special form, though, the consequent expression is only evaluated when the predicate expression is true. Hence, the expression (**if** *false* (*cdr null*) 1) evaluates to 1.

Although this example is contrived, we often take advantage of this property of the if special form in recursive definitions. Nearly every recursive procedure we define for processing lists has a base case like (**if** (*null? p*) . . .) where the alternate expression applies list-extracting procedures list (*cdr p*). If it were not for the special evaluation rule for if-expressions, the if expression would produce an error when the end of the list is reached.

With lazy evaluation, an expression is not evaluated until its value is needed. Lazy evaluation changes the evaluation rule for applications of constructed procedures. Instead of evaluating all operand expressions, lazy evaluation delays evaluation of an operand expression until the value of the parameter is needed. To keep track of what is needed to perform the evaluation when and if it is needed, a special object known as a *thunk* is created and stored in the place associated with the parameter name. By delaying evaluation of operand expressions until their value is needed, we can enable programs to define procedures that conditionally evaluate their operands, such as the *ifp* procedure.

thunk

The lazy evaluation rule for applying constructed procedures is:

> **Lazy Application Rule 2: Constructed Procedures.** To apply a constructed procedure:
>
> 1. Construct a new environment, whose parent is the environment of the applied procedure.
> 2. For each procedure parameter, create a place in the frame of the new environment with the name of the parameter. **Put a *thunk* in that place, which is an object that can be used later to evaluate the value of the corresponding operand expression if and when its value is needed.**
> 3. Evaluate the body of the procedure in the newly created environment. The resulting value is the value of the application.

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

The rule is identical to the Statefull Application Rule except for the bolded part of step 2. In the next section, we explain how to modify the Charme interpreter from the previous chapter to implement lazy evaluation. In Section 13.3, we provide some examples of programming with lazy evaluation.

Confusingly, lazy evaluation is also known as *normal order* evaluation, even though in the Scheme language it is not the normal evaluation order.[1] 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.

applicative-order

13.2 Delaying Evaluation

To implement lazy evaluation we modify the interpreter to implement the lazy application rule that delays evaluating the operand expressions until they are needed. We start by defining a Python class for representing thunks and then modify the interpreter to support lazy evaluation.

Thunks. A thunk keeps track of an expression whose evaluation is delayed until it is needed. Once the evaluation is performed, the resulting value is saved so the expression does not need to be re-evaluated the next time the value is needed.

[1] Some languages (including Haskell and Miranda) provide lazy evaluation as the standard application rule.

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).

The *Thunk* class implements thunks:

```
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
```

A *Thunk* object keeps track of the expression in the *_expr* instance variable. Since the value of the expression may be needed when the evaluator is evaluating an expression in some other environment, it also keeps track of the environment in which the thunk expression should be evaluated in the *_env* instance variable.

The *_evaluated* instance variable is a Boolean that records whether or not the thunk expression has been evaluated. Initially this value is False. After the expression is evaluated, the *_value* instance variable keeps track of its value.

The *value* method uses *forceeval* (defined in the next section) to obtain the evaluated value of the thunk expression and stores that result in the *_value* instance variable.

The *isThunk* procedure returns True only when its parameter is a thunk:

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

Delayed and forced evaluation. To implement lazy evaluation, we change the evaluator so there are two different evaluation procedures: *meval* is the standard evaluation procedure (which leaves thunks in their unevaluated state), and *forceeval* is the evaluation procedure that forces thunks to be evaluated to values. The interpreter uses *meval* when the actual expression value may not be needed, and *forceeval* to force evaluation of thunks when the value of an expression is needed.

In the *meval* procedure, a thunk evaluates to itself. We add a new **elif** clause for thunk objects to the *meval* procedure:

```
    elif isThunk(expr): # Added to support lazy evaluation
```

```
        return expr # A thunk evaluates to itself
```

The *forceeval* procedure first uses *meval* to evaluate the expression normally. If the result is a thunk, it uses the *Thunk.value* method to force evaluation of the thunk expression. The *Thunk.value* method uses *forceeval* to find the value of the thunk expression, so any thunks inside the expression will be recursively evaluated.

```
def forceeval(expr, env):
  value = meval(expr, env)
  if isThunk(value): return value.value() # force evaluation of Thunk
  else: return value
```

Next, we change the application rule to perform delayed evaluation and change a few other places in the interpreter to use *forceeval* instead of *meval* to obtain the actual values when they are needed.

Lazy applications. We change the *evalApplication* procedure to delay evaluation of the operants by creating *Thunk* objects representing each operand. Only the first subexpression (the procedure to be applied) must be evaluated. Hence, *evalApplication* uses *forceeval* to obtain the value of the first subexpression, but makes Thunk objects for the operand expressions:

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

Primitive applications. 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:

```
def mapply(proc, operands):
  def deThunk(expr):
    if isThunk(expr): return expr.value()
    else: return expr

  if (isPrimitiveProcedure(proc)):
    ops = map (deThunk, operands)
    return proc(ops)
  elif isinstance(proc, Procedure):
    ... # same as in Charme interpreter
```

Decisions. To evaluate an if-expression, it is necessary to know the actual value of the predicate expressions. We change the *evalIf* procedure to use *forceeval* when evaluating the predicate expression:

```
def evalIf(expr,env):
  if forceeval(expr[1], env) != False: return meval(expr[2],env)
  else: return meval(expr[3],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 if expression evaluates; the evaluations of the consequent and alternate expressions are left as *meval*s since it is not necessary to force them to be evaluated yet.

Printing results. The final change to the interpreter is to force evaluation when the result is displayed to the user in the *evalLoop* procedure by replacing the call to *meval* with *forceeval*.

13.3 Lazy Programming

Lazy evaluation enables programming constructs that are not possible with eager evaluation. For example, with lazy evaluation the *ifp* procedure defined at the beginning of this chapter behaves like the if-expression special form.

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.

Suppose we define 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 true)))
(define cdr (lambda (p) (p false)))

(define null false)
(define null? (lambda (x) (= x false)))
```

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

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

With eager evaluation, (*ints-from* 1) would never finish evaluating; it 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 fail to terminate even with lazy evaluation. For example, assume the standard definition of *list-length*:

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

An evaluation of (*length* (*ints-from* 1)) never terminates. Every time an application of *list-length* is evaluated, it applies *cdr* to the input list, which causes *ints-from* to evaluate another *cons*, increasing the length of the list by one. The actual length of the list is infinite, so the application of *list-length* does not terminate.

Lists with delayed evaluation can be used in useful programs. Reconsider the Fibonacci sequence from Chapter 7. Using lazy evaluation, we can define a list that is the infinitely long Fibonacci sequence:[2]

```
(define fibo-gen (lambda (a b) (cons a (fibo-gen b (+ a b)))))
(define fibos (fibo-gen 0 1))
```

The n^{th} Fibonacci number is the n^{th} element of *fibos*:

```
(define fibo (lambda (n) (list-get-element fibos n)))
```

where *list-get-element* is defined as it was defined in Chapter 5.

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.

[2]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.

```
(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 sequence 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. 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 as an infinite list using delayed evaluation.

Exercise 13.2. Describe the infinite list defined by each of the following definitions. (Check your answers by evaluating the expressions in Lazy-Charme.)

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

Exercise 13.3. [⋆⋆] A simple procedure known as the *Sieve of Eratosthenes* for finding prime numbers was created by Eratosthenes, an ancient Greek mathematician and astronomer. The procedure imagines starting with an (infinite) list of all the integers starting from 2. Then, it repeats the following two steps forever:

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

To carry out the procedure 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. But, with delayed evaluation, we can implement the Sieve procedure on an effectively infinite list.

Eratosthenes

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

13.4 Summary

We can produce new languages by changing the evaluation rules of an interpreter. Changing the evaluation rules changes what programs mean, and enables new approaches to solving problems.

Lazy evaluation increases the expressiveness of a language by supporting delayed evaluation. The main disadvantage of lazy evaluation is it makes the evaluation rules more complex. This makes it somewhat more complex to build an interpreter, but more importantly, makes it more difficult for humans to understand and reason about programs.

> *I think that there is far too much work done in the world, that immense harm is caused by the belief that work is virtuous, and that what needs to be preached in modern industrial countries is quite different from what always has been preached. Everyone knows the story of the traveler in Naples who saw twelve beggars lying in the sun (it was before the days of Mussolini), and offered a lira to the laziest of them. Eleven of them jumped up to claim it, so he gave it to the twelfth. This traveler was on the right lines. But in countries which do not enjoy Mediterranean sunshine idleness is more difficult, and a great public propaganda will be required to inaugurate it. . . . All this is only preliminary. I want to say, in all seriousness, that a great deal of harm is being done in the modern world by belief in the virtuousness of work, and that the road to happiness and prosperity lies in an organized diminution of work.*
>
> *First of all: what is work? Work is of two kinds: first, altering the position of matter at or near the earth's surface relatively to other such matter; second, telling other people to do so. The first kind is unpleasant and ill paid; the second is pleasant and highly paid. The second kind is capable of indefinite extension: there are not only those who give orders, but those who give advice as to what orders should be given. Usually two opposite kinds of advice are given simultaneously by two organized bodies of men; this is called politics.*
>
> Bertrand Russell, *In Praise of Idleness*, 1932