Lazy Evaluation

Function calls and setting of variables in Java, C++, and most other languages (including Common Lisp) are executed using an ***eager evaluation*** strategy: When calling a function, nonconstant actual arguments are evaluated *before transferring control to the called function*, and the called function is provided with the arguments' values. Similarly, code that specifies that a variable is to be set to a nonconstant expression's value is executed by immediately evaluating that expression and storing its value in the variable's location. Eager evaluation is also called ***strict*** evaluation.

Lazy evaluation is an alternative strategy that may be used for evaluating calls of programmer-defined functions and for setting variables (including fields of data objects). It is most often used in functional programming, as it can be tricky to reason correctly about the behavior of code executed using lazy evaluation when variables and data structures are not immutable. Lazy evaluation does ***not*** evaluate nonconstant arguments of a programming-defined function when calling the function, but ***delays*** evaluation of any such argument until *the first time its value is needed*. (If its value is never needed, the argument will never be evaluated.) Similarly, when executing code that specifies that a variable is to be set to a nonconstant expression's value, lazy evaluation ***delays*** evaluation of that expression: It will be evaluated *the first time the variable's value is needed*.

Languages that use lazy evaluation by default are called *lazy languages*. The functional programming language Haskell is the best-known lazy language today. Another lazy language is the Lazy Racket dialect of Scheme. Some languages that are not lazy provide features that make it easy for a programmer to write code that will evaluate particular expressions lazily; Scheme, Scala, F#, and OCaml are examples of such languages.

In pure functional programming, if eager evaluation of an expression terminates without error, then lazy evaluation of the same expression also terminates without error and returns the same value. So, since functional programmers tend to think of their code as specifying ***what*** is to be computed rather than ***how*** the computation is done, they may frequently be unconcerned with whether eager or lazy evaluation is used.

But it can be important to be know what evaluation strategy is used even in pure functional programming, as there are cases where lazy evaluation succeeds but eager evaluation fails, and there are cases where lazy evaluation is much more efficient than eager evaluation (though eager evaluation is usually more efficient when both evaluation strategies succeed). Eager evaluation fails when its evaluation of some expression produces an error (e.g., divide by 0) or does not terminate. But if the value of such an expression is not needed, then lazy evaluation will not fail for the same reason because it will never evaluate that expression. Similarly, eager evaluation may be less efficient than lazy evaluation if it evaluates a computationally expensive expression that would not be evaluated by lazy evaluation.

Lazy evaluation is usually implemented using ***memoized thunks*** of expressions; a memoized thunk is also called a ***promise***. For brevity, we will refer to memoized thunks simply as thunks. An expression's thunk is an object from which it is possible to obtain the value of the expression; ***forcing*** the thunk will obtain the expression's value and return it. An expression's thunk can be in one of two states: *unevaluated* and *evaluated*: In the *unevaluated* state the thunk contains *a parameterless function that, when called, will compute and return the expression's value*. In the *evaluated* state the thunk contains *the expression's value*.

Forcing an expression's thunk when it is in the *unevaluated* state will call the function it contains, cache the computed value of the expression in the thunk, change the thunk's state to *evaluated*, and return the computed value. Forcing an expression's thunk when it is in the *evaluated* state will just return the value that is cached in the thunk (without changing the thunk's state). So, regardless of how many times the thunk is forced, the expression's value will not be computed more than once.

Lazy evaluation of calls of programmer-defined functions and lazy setting of variables will be discussed in more depth below. The former can in fact be viewed as a special case of the latter in which the variables are the formal parameters of the programmer-defined function.

Lazy Evaluation of Calls of Programmer-Defined Functions

Lazy evaluation of a call of a programmer-defined function transfers control to the called function without evaluating any actual argument. It provides a thunk of each actual argument to the called function. In the case of nonconstant arguments, the thunk will initially be in the *unevaluated* state. An argument's thunk will be forced whenever the value of the corresponding formal parameter is needed during execution of the called function's body.

To illustrate the difference between eager and lazy evaluation, we consider the evaluation of a function call $f(g(x), h(y))$.

Let the first and second formal parameters of f be a and b, respectively. Then ***eager evaluation*** of $f(g(x), h(y))$ proceeds as follows:

1. Call $g(x)$ and $h(y)$, and provide the ***values*** of $g(x)$ and $h(y)$ to the body of f by storing the values returned by the calls of $g(x)$ and $h(y)$ in the locations of the parameters a and b.
2. Execute the body of f, which will return the value of $f(g(x), h(y))$.

Lazy evaluation of $f(g(x), h(y))$ differs from eager evaluation in the following ways:

- $g(x)$ and $h(y)$ are ***not*** called before control is transferred to the body of f. Instead of providing the *values* of $g(x)$ and $h(y)$ to the body of f (as eager evaluation would do), lazy evaluation provides *thunks* of $g(x)$ and $h(y)$. Both thunks are initially in the unevaluated state; they contain parameterless functions that, when called, will call and return the value of $g(x)$ or $h(y)$. (Note that the calls of $g(x)$ and $h(y)$ would also be lazily evaluated, so they would provide thunks of x and y to g and h. Here x and y are the variables denoted by those names at the point in the source code where the call $f(g(x), h(y))$ appears.)
- During execution of f's body, $g(x)$'s thunk is forced each time the value of parameter a is needed, and $h(y)$'s thunk is forced each time the value of parameter b is needed,

  Suppose, for example, that the body of f is:

  ```
  {
     if (a>1) return b+3;
     else return a+5;
  }
  ```

  Then, during execution of the body of f, the first time we need the value of f's first parameter a is when `(a>1)` is to be evaluated, so $g(x)$'s thunk will be forced at that time. This will call $g(x)$, cache the value of $g(x)$, change the state of $g(x)$'s thunk to *evaluated*, and return the value of $g(x)$ as a's value.

  If `(a>1)` is true, then $b + 3$ will need to be evaluated and returned. To evaluate $b + 3$ we need the value of f's second parameter b, so $h(y)$'s thunk will be forced at that time. This will call $h(y)$, cache the value of $h(y)$, change the state of $h(y)$'s thunk to *evaluated*, and return the value of $h(y)$ as b's value.

  If `(a>1)` is false, then $a + 5$ will need to be evaluated and returned, so $g(x)$'s thunk will be forced a second time to obtain the value of parameter a. But since the state of $g(x)$'s thunk is now *evaluated*, this force operation will just retrieve and return the value of $g(x)$ that was computed and cached the first time the thunk was forced: $g(x)$ will ***not*** be called again.

  Note: In the case where `(a>1)` is false, $h(y)$'s thunk is never forced since parameter b's value is never used, so $h(y)$ is never called!

Lazy Setting of Fields of Data Objects and Other Variables

In most programming languages (including Common Lisp), it is common to set the fields of a data object when that data object is created. In pure functional programming, this prevents a data object from referencing any data object that will be created later (since fields are immutable).

In contrast, when variables in a lazy language (such as fields of data objects) are to be set to values that are determined by nonconstant expressions, those values are ***not*** immediately computed. For example, when `CONS` is called in a lazy dialect of Lisp such as Lazy Racket, a nonconstant argument of the call of `CONS` will ***not*** be immediately evaluated. (Recall from sec. 2.10 of Touretzky that `CONS` is the constructor function used to create a cons cell, and that the arguments of `CONS` give the values of the car and the cdr fields of that cons cell.)

Lazy evaluation creates a thunk for each variable whose value is to be set. When the thunk is forced it will return the value that the variable is to be set to, but the thunk will be forced only when the variable's value is needed.

For example, when `CONS` is called in a lazy dialect of Lisp such as Lazy Racket, it will create and return a cons cell whose car and cdr fields contain thunks. When the program needs to use the value of that cons cell's car or cdr, it will force the corresponding thunk.

Lazy setting of variables makes it possible for variables of functional programs, such as fields of data objects, to reference data objects that will be created later if they are needed. This enables the creation of data structures that are conceptually larger than can be stored or even infinite in size, such as a list that conceptually has infinitely many elements.

Example: When its argument is an integer, the following Lazy Racket function `integers-geq` returns the ***infinite*** list of all the integers that are ≥ the argument value.

```
(define (integers-geq n)
   (cons n (integers-geq (+ n 1))))
```

Here is an illustration of how this function can be used:

```
(fourth (integers-geq 1))
   = (car (cdr (cdr (cdr (integers-geq 1)))))  =>  4
```