Laziness

greg

2002-09-27

Is Haskell eager or lazy? Consider the following interaction:

Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> (\ x -> 3) (head [])
3

From the second expression, we get the answer 3, so we know the argument, head [], has not been evaluated. We can conclude from this that Haskell is lazy.

Unfortunately, this example doesn’t give any hint of why laziness might be useful to a programmer. After all, how often do we write programs that don’t use their arguments?

Generally, we don’t write functions that ignore their arguments, but we frequently write functions that use different subsets of their arguments depending upon dynamic conditions. All programming languages provide constructs for conditional evaluation (such as if, cond, and short-circuit boolean operators), and it is often critical that these only evaluate specific subexpressions when necessary.

Lazy languages take this idea even further by universally performing computations only when necessary. This lets us make another distinction—we can have functions that just use parts of their arguments. For example, consider the function take:

\[ \text{take} :: \text{Int} \rightarrow \text{[a]} \rightarrow \text{[a]} \]
\[ \text{take} \ n \ (x:xs) = \begin{cases} \text{take} \ (n - 1) \ xs & \text{if } n > 0 \\ \text{[]} & \text{otherwise} \end{cases} \]

Take “takes” up to n elements from the beginning of a list, ignoring the rest, if any.

When might such a function be useful? For example, suppose we have a function that computes the eigenvalues of a matrix. Algorithms for this problem will naturally produce the values in decreasing order of magnitude, and in most applications, only the few largest ones are meaningful. In a lazy language, we can naturally express the computation of the complete eigensystem, and if we use \text{take} to extract the values of interest, the program will only compute the values it needs.

If we wanted the same behavior in an eager language, then we could not express the computation and selection as two distinct operations—they would have to be fused into a single “loop”, and the programmer would need to schedule the computation manually. Things would become even more complicated if we wanted to select a prefix of a list according to dynamic criteria. Haskell’s Prelude defines another function for this, takeWhile:

\[ \text{takeWhile} :: (\text{a} \rightarrow \text{Bool}) \rightarrow \text{[a]} \rightarrow \text{[a]} \]
\[ \text{takeWhile} \ p \ (x:xs) = \begin{cases} \text{takeWhile} \ p \ xs & \text{if } p \ x \\ \text{[]} & \text{otherwise} \end{cases} \]

TakeWhile takes elements from a list as long as they satisfy a given predicate. For example, in the above example, we might want to continue taking values until their magnitude drops below a certain threshold. Again, if we never use a value, the lazy program will not bother computing it.

As a result of the phenomenon just described, we can actually create infinite lists in a lazy language. As a simple example, consider the following definition:

```
ones :: [Int]
ones = 1 : ones

In Haskell, this is legal and produces an infinite (cyclic) list of 1s. As long as we don’t try to look at the whole list, we aren’t in danger of nontermination. While such a list may not seem useful at first glance, there are many common examples where it is convenient to have a simple cyclic list, and Haskell’s Prelude defines a function repeat for constructing them:¹

repeat :: a -> [a]
repeat x xs where xs = x : xs

Examples of its use often arise when we have lists of different lengths and we want to zip them together, padding the shorter list with a default value. Recall the definitions of zipWith and zip, which return a list only as long as the shorter of their arguments:

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys

zip :: [a] -> [b] -> [(a, b)]
zip = zipWith (,)

As a concrete example, suppose we’ve run a contest and need to allocate points to the winners. The top six contestants get 10, 6, 4, 3, 2, and 1 point(s), respectively, while all the others get 0. If we have a long list of contestants sorted in descending order, and we want to construct a new list where each is paired with his or her corresponding award, we can easily do this in Haskell:

awards = [10, 6, 4, 3, 2, 1]
zip contestants (awards ++ repeat 0)

Of course, we could express this computation quite easily in an eager language too, but again, laziness allows a simpler, more elegant solution.

We might also want cyclic lists that contain different elements. Haskell provides a function cycle for constructing these:

cycle :: [a] -> [a]
cycle [] = error "cycle: empty list"
cycle xs = xs ++ xss

Such lists are also useful in combination with the zip functions. For example, suppose we’re generating a large graphical table to represent a list of items. To improve readability, we want to change the color between each row and the next. In Haskell, we can write something like:

alternateColors :: [Item] -> [ColoredItem]
alternateColors items = zipWith colorize items
  (cycle [LightGreen, LightBlue, Yellow])

Another application of this idiom is in music. There is a collection of Haskell datatypes and combinators, called Haskore, that support composition and interpretation of music. One thing it allows is to define (infinitely) repeating phrases, which the user can compose in parallel with a sequence of melodic phrases. As with zip, this composition takes only enough of the infinite structure to match the finite one. This capability is highly convenient, since the user can make a change in one place, and the accompanying parts will adapt automatically.

In Haskell, we can even construct infinite, non-cyclic lists. For example, the following two definitions construct the infinite list of natural numbers and the infinite list of Fibonacci numbers. Of course, as with cyclic lists, Haskell only computes elements when they are needed.

¹We might initially think of using a simpler definition like repeat x = x : repeat x. Can you see why Haskell doesn’t use this definition?
nats = 0 : map succ nats
fib = 1 : 1 : zipWith (+) fib (tail fib)

In many cases, laziness allows us not to worry about when or even if a list terminates. Furthermore, it eliminates
the distinction between a data structure and a computation that builds a data structure. We can use this idea to define
stream-processors in Haskell very naturally. For example, consider modeling a boolean circuit. We can easily define
basic logic gates as list functions:

notGate = map not
andGate = zipWith (&&)
orGate = zipWith (||)

We can construct an XOR gate by composing these gates:

xorGate l1 l2 = out
    where out = orGate (andGate l1 (notGate l2))
          (andGate l2 (notGate l1))

If we want to convert this gate into a flip-flop by feeding the output back in through a delay, we can easily do so:

delayGate = (False :)

flipFlop inp = out
    where out = orGate (andGate inp (notGate feedBack))
         (andGate feedBack (notGate inp))
    feedBack = delayGate out

Without laziness, the situation would be much more complicated. We would not be able to use the convenient list
abstraction, and we would need to maintain state and schedule operations manually.

We saw earlier how we could implement recursion with cyclic data structures, but in an eager language, this
required impure features—sequencing and mutation. Lazy languages allow us to create cyclic structures without any
such issues. The following Haskell code interprets a recursive expression by constructing a cyclic environment:

interp (Rec name exp body) env = interp body recEnv
where recEnv = aSub name (interp exp recEnv) env