Lazy evaluation

CIS 194 Week 6
18 February 2012

Suggested reading:

- **foldr foldl foldl'** from the Haskell wiki

On the first day of class I mentioned that Haskell is *lazy*, and promised to eventually explain in more detail what this means. The time has come!

Strict evaluation

Before we talk about *lazy evaluation* it will be useful to look at some examples of its opposite, *strict evaluation*.

Under a strict evaluation strategy, function arguments are completely evaluated *before* passing them to the function. For example, suppose we have defined

```
f x y = x + 2
```

In a strict language, evaluating `f 5 (29^35792)` will first completely evaluate `5` (already done) and `29^35792` (which is a lot of work) before passing the results to `f`.

Of course, in this *particular* example, this is silly, since `f` ignores its second argument, so all the work to compute `29^35792` was wasted. So why would we want this?

The benefit of strict evaluation is that it is easy to predict *when* and *in what order* things will happen. Usually languages with strict evaluation will even specify the order in which function arguments should be evaluated (*e.g.* from left to right).

For example, in Java if we write

```
f (release_monkeys(), increment_counter())
```

we know that the monkeys will be released, and then the counter will be incremented, and then the results of doing those things will be passed to `f`, and it does not matter whether `f` actually ends up using those results.

If the releasing of monkeys and incrementing of the counter could independently happen, or not, in either order, depending on whether `f` happens to use their results, it would be extremely confusing. When such "side effects" are allowed, strict evaluation is really what you want.

Side effects and purity

So, what's really at issue here is the presence or absence of *side effects*. By "side effect" we mean *anything that causes evaluation of an expression to interact with something outside itself*. The root issue is that such outside interactions are time-sensitive. For example:

- Modifying a global variable — it matters when this happens since it may affect the evaluation of other expressions
- Printing to the screen — it matters when this happens since it may need to be in a certain order with respect to other writes to the screen
- Reading from a file or the network — it matters when this happens since the contents of the file can affect the outcome of the expression

As we have seen, lazy evaluation makes it hard to reason about when things will be evaluated; hence including

side effects in a lazy language would be extremely unintuitive. Historically, this is the reason Haskell is pure: initially, the designers of Haskell wanted to make a *lazy* functional language, and quickly realized it would be impossible unless it also disallowed side effects.

But… a language with *no* side effects would not be very useful. The only thing you could do with such a language would be to load up your programs in an interpreter and evaluate expressions. (Hmm… that sounds familiar…) You would not be able to get any input from the user, or print anything to the screen, or read from a file. The challenge facing the Haskell designers was to come up with a way to allow such effects in a principled, restricted way that did not interfere with the essential purity of the language. They finally did come up with something (namely, the `IO` monad) which we'll talk about in a few weeks.

Lazy evaluation

So now that we understand strict evaluation, let's see what lazy evaluation actually looks like. Under a lazy evaluation strategy, evaluation of function arguments is *delayed as long as possible*: they are not evaluated until it actually becomes necessary to do so. When some expression is given as an argument to a function, it is simply packaged up as an *unevaluated expression* (called a "thunk", don't ask me why) without doing any actual work.

For example, when evaluating `f 5 (29^35792)`, the second argument will simply be packaged up into a thunk without doing any actual computation, and `f` will be called immediately. Since `f` never uses its second argument the thunk will just be thrown away by the garbage collector.

Pattern matching drives evaluation

So, when is it "necessary" to evaluate an expression? The examples above concentrated on whether a function *used* its arguments, but this is actually not the most important distinction. Consider the following examples:

```
f1 :: Maybe a -> [Maybe a]
f1 m = [m,m]

f2 :: Maybe a -> [a]
f2 Nothing  = []
f2 (Just x) = [x]
```

`f1` and `f2` both *use* their argument. But there is still a big difference between them. Although `f1` uses its argument `m`, it does not need to know anything about it. `m` can remain completely unevaluated, and the unevaluated expression is simply put in a list. Put another way, the result of `f1 e` does not depend on the shape of `e`.

`f2`, on the other hand, needs to know something about its argument in order to proceed: was it constructed with `Nothing` or `Just`? That is, in order to evaluate `f2 e`, we must first evaluate `e`, because the result of `f2` depends on the shape of `e`.

The other important thing to note is that thunks are evaluated *only enough* to allow a pattern match to proceed, and no further! For example, suppose we wanted to evaluate `f2 (safeHead [3^500, 49])`. `f2` would force evaluation of the call to `safeHead [3^500, 49]`, which would evaluate to `Just (3^500)`—note that the `3^500` is *not* evaluated, since `safeHead` does not need to look at it, and neither does `f2`. Whether the `3^500` gets evaluated later depends on how the result of `f2` is used.

The slogan to remember is "*pattern matching drives evaluation*". To reiterate the important points:

- Expressions are only evaluated when pattern-matched
- …only as far as necessary for the match to proceed, and no farther!

Let's do a slightly more interesting example: we'll evaluate `take 3 (repeat 7)`. For reference, here are the definitions of `repeat` and `take`:

```
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _       | n <= 0 =  []
take _ []               =  []
take n (x:xs)           =  x : take (n-1) xs
```

Carrying out the evaluation step-by-step looks something like this:

```
  take 3 (repeat 7)
      { 3 <= 0 is False, so we proceed to the second clause, which
    needs to match on the second argument. So we must expand
    repeat 7 one step. }
= take 3 (7 : repeat 7)
      { the second clause does not match but the third clause
        does. Note that (3-1) does not get evaluated yet! }
= 7 : take (3-1) (repeat 7)
      { In order to decide on the first clause, we must test (3-1)
        <= 0 which requires evaluating (3-1). }
= 7 : take 2 (repeat 7)
      { 2 <= 0 is False, so we must expand repeat 7 again. }
= 7 : take 2 (7 : repeat 7)
      { The rest is similar. }
= 7 : 7 : take (2-1) (repeat 7)
= 7 : 7 : take 1 (repeat 7)
= 7 : 7 : take 1 (7 : repeat 7)
= 7 : 7 : 7 : take (1-1) (repeat 7)
= 7 : 7 : 7 : take 0 (repeat 7)
= 7 : 7 : 7 : []
```

(Note that although evaluation *could* be implemented exactly like the above, most Haskell compilers will do something a bit more sophisticated. In particular, GHC uses a technique called *graph reduction*, where the expression being evaluated is actually represented as a *graph*, so that different parts of the expression can share pointers to the same subexpression. This ensures that work is not duplicated unnecessarily. For example, if `f x = [x,x]`, evaluating `f (1+1)` will only do *one* addition, because the subexpression `1+1` will be shared between the two occurrences of `x`.)

Consequences

Laziness has some very interesting, pervasive, and nonobvious consequences. Let's explore a few of them.

Purity

As we've already seen, choosing a lazy evaluation strategy essentially *forces* you to also choose purity (assuming you don't want programmers to go insane).

Understanding space usage

Laziness is not all roses. One of the downsides is that it sometimes becomes tricky to reason about the space usage of your programs. Consider the following (innocuous-seeming) example:

```
-- Standard library function foldl, provided for reference
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Let's consider how evaluation proceeds when we evaluate `foldl (+) 0 [1,2,3]` (which sums the numbers in a list):

```
  foldl (+) 0 [1,2,3]
= foldl (+) (0+1) [2,3]
= foldl (+) ((0+1)+2) [3]
```

```
= foldl (+) (((0+1)+2)+3) []
= (((0+1)+2)+3)
= ((1+2)+3)
= (3+3)
= 6
```

Since the value of the accumulator is not demanded until recursing through the entire list, the accumulator simply builds up a big unevaluated expression `(((0+1)+2)+3)`, which finally gets reduced to a value at the end. There are at least two problems with this. One is that it's simply inefficient: there's no point in transferring all the numbers from the list into a different list-like thing (the accumulator thunk) before actually adding them up. The second problem is more subtle, and more insidious: evaluating the expression `(((0+1)+2)+3)` actually requires pushing the `3` and `2` onto a stack before being able to compute `0+1` and then unwinding the stack, adding along the way. This is not a problem for this small example, but for very long lists it's a big problem: there is usually not as much space available for the stack, so this can lead to a stack overflow.

The solution in this case is to use the `foldl'` function instead of `foldl`, which adds a bit of strictness: in particular, `foldl'` requires its second argument (the accumulator) to be evaluated before it proceeds, so a large thunk never builds up:

```
  foldl' (+) 0 [1,2,3]
= foldl' (+) (0+1) [2,3]
= foldl' (+) 1 [2,3]
= foldl' (+) (1+2) [3]
= foldl' (+) 3 [3]
= foldl' (+) (3+3) []
= foldl' (+) 6 []
= 6
```

As you can see, `foldl'` does the additions along the way, which is what we really want. But the point is that in this case laziness got in the way and we had to make our program *less* lazy.

(If you're interested in learning about *how* `foldl'` achieves this, you can **read about `seq` on the Haskell wiki**.)

Short-circuiting operators

In some languages (Java, C++) the boolean operators `&&` and `||` (logical AND and OR) are *short-circuiting*: for example, if the first argument to `&&` evaluates to false, the whole expression will immediately evaluate to false without touching the second argument. However, this behavior has to be wired into the Java and C++ language standards as a special case. Normally, in a strict langauge, both arguments of a two-argument function are be evaluated before calling the function. So the short-circuiting behavior of `&&` and `||` is a special exception to the usual strict semantics of the language.

In Haskell, however, we can define short-circuiting operators without any special cases. In fact, `(&&)` and `(||)` are just plan old library functions! For example, here's how `(&&)` is defined:

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && _ = False
```

Notice how this definition of `(&&)` does not pattern-match on its second argument. Moreover, if the first argument is `False`, the second argument is entirely ignored. Since `(&&)` does not pattern-match on its second argument at all, it is short-circuiting in exactly the same way as the `&&` operator in Java or C++.

Notice that `(&&)` also could have been defined like this:

```
(&&!) :: Bool -> Bool -> Bool
True  &&! True  = True
True  &&! False = False
False &&! True  = False
False &&! False = False
```

While this version takes on the same values as `(&&)`, it has different behavior. For example, consider the following:

```
False &&  (34^9784346 > 34987345)
False &&! (34^9784346 > 34987345)
```

These will both evaluate to `False`, but the second one will take a whole lot longer! Or how about this:

```
False &&  (head [] == 'x')
False &&! (head [] == 'x')
```

The first one is again `False`, whereas the second one will crash. Try it!

All of this points out that there are some interesting issues surrounding laziness to be considered when defining a function.

User-defined control structures

Taking the idea of short-circuiting operators one step further, in Haskell we can define our own *control structures*.

Most languages have some sort of special built-in `if` construct. Some thought reveals why: in a way similar to short-circuiting Boolean operators, `if` has special behavior. Based on the value of the test, it executes/evaluates only *one* of the two branches. It would defeat the whole purpose if both branches were evaluated every time!

In Haskell, however, we can define `if` as a library function!

```
if' :: Bool -> a -> a -> a
if' True  x _ = x
if' False _ y = y
```

Of course, Haskell *does* have special built-in `if`-expressions, but I have never quite understood why. Perhaps it is simply because the language designers thought people would expect it. "What do you mean, this language doesn't have `if`!?" In any case, `if` doesn't get used that much in Haskell anyway; in most situations we prefer pattern-matching or guards.

We can also define other control structures—we'll see other examples when we discuss monads.

Infinite data structures

Lazy evaluation also means that we can work with *infinite data structures*. In fact, we've already seen a few examples, such as `repeat 7`, which represents an infinite list containing nothing but `7`. Defining an infinite data structure actually only creates a thunk, which we can think of as a "seed" out of which the entire data structure can *potentially* grow, depending on what parts actually are used/needed.

Another practical application area is "effectively infinite" data structures, such as the trees that might arise as the state space of a game (such as go or chess). Although the tree is finite in theory, it is so large as to be effectively infinite—it certainly would not fit in memory. Using Haskell, we can define the tree of all possible moves, and then write a separate algorithm to explore the tree in whatever way we want. Only the parts of the tree which are actually explored will be computed.

Pipelining/wholemeal programming

As I have mentioned before, doing "pipelined" incremental transformations of a large data structure can actually be memory-efficient. Now we can see why: due to laziness, each stage of the pipeline can operate in lockstep, only generating each bit of the result as it is demanded by the next stage in the pipeline.

Dynamic programming

As a more specific example of the cool things lazy evaluation buys us, consider the technique of ***dynamic programming***. Usually, one must take great care to fill in entries of a dynamic programming table in the proper order, so that every time we compute the value of a cell, its dependencies have already been computed. If we get the order wrong, we get bogus results.

However, using lazy evaluation we can get the Haskell runtime to work out the proper order of evaluation for us! For example, here is some Haskell code to solve the **0-1 knapsack problem**. Note how we simply define the array `m` in terms of itself, using the standard recurrence, and let lazy evaluation work out the proper order in which to compute its cells.

```
import Data.Array

knapsack01 :: [Double]   -- values
           -> [Integer]  -- nonnegative weights
```

```haskell
           -> Integer     -- knapsack size
           -> Double      -- max possible value
knapsack01 vs ws maxW = m!(numItems-1, maxW)
  where numItems = length vs
        m = array ((-1,0), (numItems-1, maxW)) $
              [((-1,w), 0) | w <- [0 .. maxW]] ++
              [((i,0), 0) | i <- [0 .. numItems-1]] ++
              [((i,w), best)
                  | i <- [0 .. numItems-1]
                  , w <- [1 .. maxW]
                  , let best
                          | ws!!i > w  = m!(i-1, w)
                          | otherwise = max (m!(i-1, w))
                                            (m!(i-1, w - ws!!i) + vs!!i)
              ]

example = knapsack01 [3,4,5,8,10] [2,3,4,5,9] 20
```

Generated 2013-03-14 14:39:59.207989