```
{-# LANGUAGE BangPatterns #-}
```

Being Lazy with Class

CIS 194 Week 7
16 October 2014

Suggested reading:

- **Real World Haskell, Chapter 25: Profiling and Optimization**

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` type).

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:

```
badSum :: Num a => [a] -> a
badSum []     = 0
badSum (x:xs) = x + badSum xs
```

`badSum` is not tail recursive. It works right to left in the list, and GHC must remember all recurrences of `badSum` on the stack, making it fail for large lists.

```
lazySum :: Num a => [a] -> a
lazySum = go 0
  where go acc []     = acc
        go acc (x:xs) = go (x + acc) xs
```

`lazySum` *is* tail recursive, but it's too lazy. The problem is that all those uses of `(+)` never get evaluated, until the very end. So, GHC just makes bigger and bigger thunks until the function is done running.

Let's watch `lazySum` work:

```
lazySum [1,2,3,4]
go 0 [1,2,3,4]
go (1 + 0) [2,3,4]
go (2 + (1 + 0)) [3,4]
go (3 + (2 + (1 + 0))) [4]
go (4 + (3 + (2 + (1 + 0)))) []
(4 + (3 + (2 + (1 + 0))))
(4 + (3 + (2 + 1)))
(4 + (3 + 3))
(4 + 6)
10
```

We need to add strictness, to avoid all those additions from accumulating!

```
strictSum :: Num a => [a] -> a
strictSum = go 0
  where go acc []     = acc
        go acc (x:xs) = acc `seq` go (x + acc) xs
```

The `seq` function has type `a -> b -> b` and *forces* the value in its first parameter before returning its second parameter. `seq` is magical, in that it can't be written in pure Haskell.

In `strictSum`, we make sure that the accumulated value `acc` is evaluated before proceeding. This eliminates all addition thunks!

Let's watch `strictSum` work:

```
strictSum [1,2,3,4]
go 0 [1,2,3,4]
go (1 + 0) [2,3,4]
go (2 + 1) [3,4]
go (3 + 3) [4]
go (4 + 6) []
(4 + 6)
10
```

That's much better!

Another way to write `strictSum` is using the `BangPatterns` language extension, enabled at the top of this file:

```
strictSum' :: Num a => [a] -> a
strictSum' = go 0
  where go  acc []     = acc
        go !acc (x:xs) = go (x + acc) xs
```

Note the `!` before `acc` in the second equation for `go`. Just like `seq`, this forces `acc` to be evaluated.

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.

Infinite lists, in particular, are quite common. For example, here is a function that pairs a list with indices into that list:

```
withIndices :: [a] -> [(a,Integer)]
withIndices xs = zip xs [0..]
```

Here, `[0..]` is the infinite list of numbers counting up from 0. We could also define that list this way:

```
nats :: [Integer]
nats = 0 : map (+1) nats
```

Strange, but true.

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.

Profiling

Thinking back to `lazySum` and `strictSum`, it is sometimes the case that laziness (or other programming decisions) lead to unfortunate consequences at runtime. Before blindly adding exclamation points to your code, it is helpful to *profile* your program to see what is going on.

To profile a program, it will first need a `main` – the actions taken by `main` are what is profiled. Then, compile it with the `-rtsopts` option, which enables extra arguments to be passed into your built program. When running a program built with `-rtsopts`, you can pass the `+RTS` option; every option you pass after that goes straight to the runtime system, not your program. These options can control profiling. We'll be using three such options:

- `-s` makes your program dump out memory and time usage information when it is done running.
- `-h` makes your program produce a *heap profile*. A heap profile details how much memory your program takes up over time. This profile is left in the file `YourProgram.hp` where your source file is named `YourProgram.hs` and your compiled program is named `YourProgram` (or `YourProgram.exe`, on Windows).
- `-i` allows you to set the heap profiling interval, that is, how often the heap is profiled. If your heap profile is empty, you may need to shorten the interval. The `-i` is followed by a the number of seconds (it is often a decimal) between heap checks. I've had success with `-i0.001`. The default is `0.1`, which is too long for the small programs we'll be testing.

After a heap profile is created, it needs to be converted into a viewable format. This is done with the `hp2ps` utility. That separate program takes many options, but we'll always use `-c`, which enables color. You call `hp2ps` on your `.hp` file, and it creates a `.ps` file, which can be opened in just about any PDF viewer.

Let's put this all into practice. I want a heap profile of `lazySum` called on the numbers 0 through 1,000,000:

```
main = print (lazySum [1..1000000])
```

Say this file is called `Lec07.hs`. Then, you will do this:

```
ghc Lec07.hs -rtsopts
./Lec07 +RTS -s -h -i0.001
hp2ps -c Lec07.hp
```

(Or, from the Windows command line, `cmd.exe`:

```
ghc Lec07.hs -rtsopts
Lec07.exe +RTS -s -h -i0.001
```

```
hp2ps -c Lec07.hp
```

)

When your program runs, you will see the memory and time information. Now, open up `Lec07.ps` and see your heap!

Note: GHC likes to be lazy and won't compile programs that have recently been compiled, without changes. This is true even if you change the compilation options, such as adding `-rtsopts`. So, if you have compiled a program and wish to enable `-rtsopts`, make sure to also pass `-fforce-recomp`, which gets GHC to actually do what you ask! (Or, you could just edit your source file.)

At this point, we're not going to worry too much about processing a heap profile. However, it's an important tool to know about in your tool chest in case you run into problems in the future. The **GHC Manual** details the runtime system options and profiling in good detail.

```
Generated 2015-02-23 10:08:43.175284
```