Haskell: Expression Evaluation

Assignment 1

It exists.

Lazy Evaluation

Consider the evaluation of the expression `fst (1+2, 3+4)`. One of these two evaluation strategies must happen:

```
fst (1+2, 3+4)
== fst (3, 3+4)
== fst (3, 7)
== 3
```

```
fst (1+2, 3+4)
== 1+2
== 3
```

Basically: do we have to evaluate `3+4` or not?

In the equivalent C, Python, etc, the answer is clear: `3+4` gets evaluated.

In Haskell, we can try giving an infinite list as the second argument and confirm that it does **not** get evaluated.

GHCI
```
Prelude> fst (1+2, 3+4)
3
Prelude> fst (1+2, [1..])
3
```

Haskell is *lazy*: it delays evaluation of any calculation as long as possible.

Whenever you use a "calculated" value, you are really just passing around a reference to the calculation that will create it.

When Haskell actually needs the value (e.g. has to display it on the screen), it starts working through the calculation, doing just enough work to get the result.

e.g. consider an operation on an infinite list, like `take 2 [1..]`. The `take` function is defined something like:

.hs
```
take 0 _ = []
take n (x:xs) = x : take (n-1) xs
```

The calculation is evaluated like this:

```
take 2 [1..]
== take 2 (1:[2..])            -- arg 2 matches x:xs?
== 1 : take (2-1) [2..]        -- recursive case
== 1 : take 1 [2..]            -- arg 1 matches 0?
== 1 : take 1 (2:[3..])        -- arg 2 matches x:xs?
== 1 : 2 : take (1-1) [3..]    -- recursive case
== 1 : 2 : take 0 [3..]        -- arg 1 matches 0?
== 1 : 2 : [] == [1, 2]        -- base case
```

Lazy evaluation means that we can define huge/complicated values, but if we only need small parts, Haskell will only calculate that.

```
*Main> let bigResults = [bigCalculation i | i<-[1..100000]]
*Main> take 3 bigResults
[1,16,7625597484987]
*Main> bigResults !! 5
265911977215322677968248940438791859490534220026…
```
GHCI

e.g. you might do this in assignment 1: define infinite result sets, but extract the ones you want.

A not-yet-evaluated expression is called a *thunk*. Haskell generates and stores thunks as needed. When a value is needed, its thunk resumes evaluation.

Thunks that don't need to be evaluated get discarded, like `[3..]` in the previous example.

[I assume "thunk" is the past-particple of the verb "to think" in some alternate universe.] Haskell wiki: Thunk

This can lead to calculations actually being done at unexpected times.

```
*Main> let x = bigCalculation 1000 -- returns immediately
*Main> let y = x+1                 -- returns immediately
*Main> y                           -- takes time
```
GHCI

Be careful when you think "that was really fast": maybe it didn't actually happen.

[This is more evidence that things like `x` and `y` here aren't "variables" as we know the concept from other languages: they don't hold the value, but the calculation.]

Controlling Laziness

Lazy evaluation can be handy, but too much lazy evaluation can be a problem. It's wasteful to store many thunks in memory if we definitely need them evaluated.

Consider `foldl`, which is lazily evaluated like this:

```
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
```

None of the addition gets done until we actually display the value (or use it in some other way). Many thunks are (invisibly) stored until then.

This is often bad for `foldl`, so there is a non-lazy version, `foldl'` in the `Data.List` module:

```
Prelude> foldl (+) 0 [1..100000000]
*** Exception: stack overflow
Prelude> import Data.List

Prelude Data.List> foldl' (+) 0 [1..100000000]
5000000050000000
```
GHCI

The evaluation of `foldl'` will be more like this:

```
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
```

There are less thunks stored with `foldl'`, so it's more efficient *if you actually need all of the results*.

If the fold produced a list and you only need *some* elements, then `foldl'` could be less efficient because it does unnecessary calculations.

tl;dr: In Haskell you have the choice of which things get calculated. Now you have to make the choice.

The `$!` operator can be used to force strict evaluation of an argument. This is most useful in recursive calls where you *know* you'll need the value, but would get a lot of thunks otherwise.

```hs
myPowerTailRec, myPowerTailStrict :: Int -> Int -> Int -> Int
myPowerTailRec a _ 0 = a
myPowerTailRec a x y = myPowerTailRec (x*a) x (y-1)

myPowerTailStrict a _ 0 = a
myPowerTailStrict a x y = (myPowerTailStrict $! (a*x)) x (y-1)
```

These are identical but the `$!` forces `a*x` to be calculated strictly.

Compare: same calculations but thunk built or not.

```
myPowerTailRec 1 2 2
== myPowerTailRec (2*1) 2 (2-1)       -- recursive case
== myPowerTailRec (2*1) 2 1           -- arg 3 matches 0?
== myPowerTailRec (2*(2*1)) 2 (1-1)   -- recursive case
== myPowerTailRec (2*(2*1)) 2 0       -- arg 3 matches 0?
== 2*(2*1)                            -- base case
== 2*2                                -- evaluate thunk
== 4                                  -- evaluate thunk
```

```
myPowerTailStrict 1 2 2
== myPowerTailStrict $! (2*1) 2 (2-1) -- recursive case
== myPowerTailStrict 2 2 (2-1)        -- strict eval of arg 1
== myPowerTailStrict 2 2 1            -- arg 3 matches 0?
== myPowerTailStrict $! (2*2) 2 (1-1) -- recursive case
== myPowerTailStrict 4 2 (1-1)        -- strict eval of arg 1
== myPowerTailStrict 4 2 0            -- arg 3 matches 0?
== 4                                  -- base case
```

The `seq` function can also control lazy evaluation. Calling...

```hs
seq a b
```

... returns `b`, but forces strict evaluation of `a`. It can be used like this to force a `let`/`where` value to be strictly evaluated:

```hs
myPowerSeq a _ 0 = a
myPowerSeq a x y = seq newacc (myPowerSeq newacc x (y-1))
    where newacc = a*x
```

Like any other optimization, `$!` and `seq` should only be used where needed. i.e. you tried something without them; it was slow **because of laziness**; being less lazy helps.

GHC's optimizer can often insert strict evaluation automatically where it's helpful.

We can try it with [many versions of the dumb power calculator](). Greg's script to see what's happening:

```sh
ulimit -v 4194304  # limit everything to 4GB memory
rm *.hi *.o strict; ghc -O0 strict.hs
./strict 5000000   # all succeed
./strict 50000000  # -Tail out of memory
rm *.hi *.o strict; ghc -O2 strict.hs
./strict 50000000  # -Tail, -Strict, -Foldl, -Foldl' ~same
./strict 500000000 # myPower, -Foldr, -Foldl out of memory
```

Function Application

Aside: The `($!)` is the strictly-evaluated sibling of `($)` which is function application, but lazy. Just like every other function call we've ever done.

Why? The `($)` has a difference precedence and association than passing an argument to a function. That can make big function compositions more readable.

[Function composition and $ operator in Haskell]; [Function application with $]

e.g. Suppose we're using `map` and `divisors` and `(*)` together like this (for some reason):

```hs
funnyDivisors n   = map pred (divisors (n*2))
```

That's a lot of parens. This is equivalent:

```hs
funnyDivisors' n  = map pred $ divisors $ n*2
```

The `($)` style is often more readable (but right-to-left) when you're combining multiple functions to get something done.

Note: `($)` combines a function and an argument; `(.)` combines two functions. All of these are the same function:

```hs
funnyDivisors n   = map pred (divisors (n*2))
funnyDivisors' n  = map pred $ divisors $ n*2
funnyDivisors'' n = (map pred) . divisors . (*2) $ n
funnyDivisors'''  = (map pred) . divisors . (*2)
```

Which is easier to understand (quickly and correctly)?

The `($!)` is just the strict version of this idea.

Pure Functions

All functions we have seen in Haskell have been *pure functions*. That is:

1. Given the same arguments, they always return the same results.
2. They have no side effects: do not modify the external state of the program.

#2 was easy: there are no variables to modify.

#1 was a slight lie: function results depend on their arguments and *free variables*. A free variable is any value used that isn't a function argument.

e.g. `foldl` is free here; `xs` is *bound*.

```hs
mySum xs = foldl (+) 0 xs
```

e.g. in Assignment 1, `pwReduce` has `pwLength` free.

Since free variables cannot be modified (i.e. are actually constants), we can still claim these are pure functions.

i.e. if `pwLength` is defined a certain way, it will be the same throughout the whole execution.

Insisting on pure functions makes anything with a state difficult. All of these depend on external state:

- file I/O: file contents on disk;
- user I/O: results of typing/clicking;
- random numbers: random seed set by hard-to-predict external inputs.

Functions doing these things must all be non-pure: results inherently depend on more than just the arguments.

Doing non-pure things in Haskell requires monads. [More later.]

Purity is what allows lazy evaluation. Haskell can choose to calculate a value or not since there are no side effects to worry about.

No side effects means there's no reason to evaluate code unless we *really* need the return value.

Concurrent Programming

Concurrent program: several things can be happening independently.

Concurrent programming is hard in most programming languages: threads, semaphores, mutexes, deadlock, ...

But the hard part is usually managing the **shared state**: shared variables must be locked and unlocked when used.

With pure functions in Haskell, there's no state to share. It should be easy to do things concurrently.

Sadly, because of lazy evaluation, it won't be completely automatic. Haskell always calculates the **one** value it needs next: it won't do multiple things because it's so aggressively lazy.

There are functions in `Control.Parallel` and `Control.Parallel.Strategies` to help you specify what can run concurrently.

We need to express: how lazy/strict we will be, and what should be done concurrently?

Recall `seq a b`: strictly evaluate `a` and return `b` (but *maybe* calculating `b` first). For concurrent execution, we have:

- `pseq a b`: like `seq` but evaluates `a` *before* returning `b`.

- `par a b`: evaluate `a` and `b` concurrently, and return `b`.

Typical usage: use these to express "do A and B in parallel and then combine them".

Suppose we have two values that take a long (but similar) time to compute:

```hs
calcA = a + b
    where a = calculation 1
          b = calculation 2
```

We can compute them in concurrently before adding:

```hs
calcB = (a `par` b) `pseq` (a + b)
    where a = calculation 1
          b = calculation 2
```

It should use two processor cores and take about half as long.

GHC has to be asked to allow parallel execution, something like:

```sh
ghc -O2 -with-rtsopts="-N8" -threaded concurrent1.hs
```

A higher-level option: `parMap` from `Control.Parallel.Strategies`. It's like `map` but will evaluate in parallel.

... but you have to specify *how* to evaluate each element. An ***execution strategy*** specifies how non-lazy to be.

For example, this will be single-threaded:

```hs
calcC = map slowCalc [0..100]
```

This is equivalent but can use several cores:

```hs
calcD = parMap rseq slowCalc [0..100]
```

[Complete code for these examples.]

It's still not trivial to make fast concurrent code.

We have to defeat Haskell's lazy evaluation.

Breaking the problem into chunks that are too small causes overhead coordinating threads. e.g. these take about the same amount of time to complete:

```hs
calcE = map fastCalc [0..1000000]
calcF = parMap rseq fastCalc [0..1000000]
```

Even if things can be done in parallel easily, there are still some decisions to be made. In particular: what size chunks of work should be done in parallel?

Too small: too much overhead starting/stopping/communicating. Too big: not enough parallelism.

See the Concurrent Fibonacci example for an exploration.

See also *Real World Haskell*, ch. 24.

Monads

Since Haskell is non-imperative and lazy, it's usually not clear in what order (or if) code will actually be evaluated.

Sometimes that's nice: can work with infinite data structures; can be expressive in interesting ways.

Sometimes it's painful: controling thunks; writing concurrent code.

Sometimes it's a disaster: reading parts of a file (non-pure functions; out-of-order evaluation could give incorrect results).

Monads give you a way to think of your code in sequence (but usual lazy evaluation rules can apply, depending on the monad).

A *monad* is basically a wrapper to a type.

An example: the `IO` monad wraps a type to indicate that its value has something to do with input or output.

The `getLine` function reads a line of input from the user and has a type that indicates it gives an string with an input/output origin:

```hs
getLine :: IO String
```

An `IO String` is a string, but wrapped so we know that the function might return a different value next time it's called.

Another example: the `Maybe` monad wraps a type but allows representation of failure.

In exercise 4, the `findElt` function returns a `Maybe Int`: either the position in a list or failure.

[Technically we're not using the monad features of `Maybe` in either Exercise 4 or Assignment 1, but it is one.]

A monad must be able to do a few operations on its values:

- Wrap and unwrap its values (e.g. convert between `2` and `Just 2`)
- Chain together operations from one monad value to another.

[There are some others, but these will get us by.]

The chaining of operations is done by the `>>=` operator (think of it as “and then”):

```hs
(>>=) :: m a -> (a -> m b) -> m b
```

1. It takes the result of the previous “step” (of type `m a`);
2. unwraps it automatically (to type `a`);
3. a function takes the unwrapped value and produces the result of this step (a function `a -> m b`);
4. The `m b` is the result of this step.

There's nothing inherently imperative about what `(>>=)` does: it's all still functional operations.

But the programmer can think in “steps” and express imperative thoughts. A specific monad implementation can implement the operations in whatever way is correct. (e.g. either strictly evaluate to force execution order, or not.)

Writing with the `(>>=)` is painful, so there's a shortcut with the `do` syntax. This code reads two lines of input, and returns the second:

```hs
secondLine :: IO String
secondLine = do
    line1 <- getLine
    line2 <- getLine
    return line2
```

That is exactly equivalent to:

```hs
secondLine' :: IO String
secondLine' =
  getLine >>= (\line1 -> getLine >>= (\line2 -> return line2))
```

Here, the result of the first `getLine` becomes an argument to the rest of the code named `line1`; same with `line2`.

The `return` function wraps a non-monad value in the appropriate monad. Here, it transforms `line2 :: String` to an `IO String`.

I think `return` is named wrong: it doesn't stop a function's execution and indicate the result (like in every other language where you've seen the word). It wraps a value up in a monad.

If `return` was instead named `wrapInMonad`, people would understand it.

Doing "`y <- f x`" takes the monad-wrapped result of `f x`, unwraps it and calls it `y` as an argument to the following code.

And `return` wraps a non-monad value in a monad. So doing `y <- f x` followed by `return y` is pointless. This is equivalent:

```hs
secondLine'' :: IO String
secondLine'' = do
    line1 <- getLine
    getLine
```

Your job when writing monadic code:

- Each "statement" is a function that takes the unwrapped values and produces a monad value.
- The last "statement" is the result; `return` can be used to wrap a non-monad value if necessary.
- If you have a step that produces a non-monad value, you can use `let` as you would anywhere else.

e.g. read a line of input and **print** the result of `map succ`.

```hs
succInput :: IO ()
succInput = do
    text <- getLine
    let succtext = map succ text
    putStrLn succtext
```

`getLine` returns `IO String`; `text` is that unwrapped to a `String`; `succtext` is built by a normal Haskell function call and is a `String`; `putStrLn` takes a `String`, does IO, and returns nothing.

`getLine` returns a monad so use `<-`, but `map` returns a non-monad so `<-` doesn't make sense.

Or we could use the wrapping of `return` and unwrapping of `<-` to make the code more uniform:

```hs
succInput' :: IO ()
succInput' = do
    text <- getLine
    succtext <- return $ map succ text
    putStrLn succtext
```

... which may be silly, but the result is the same.

Summary:

- You can think imperatively, but Haskell can still be very functional.
- A monad knows how to chain the operations together.
- Values can be wrapped/unwrapped in monads: the monad takes care of it.

Learn You A Haskell: Monads; Haskell Wikibook: Understanding monads

Computerphile: What is a Monad?

Monad: Random Numbers

A function that generates random numbers can't be pure: it must give a different result each time it's called. There is typically some internal state that keeps track of a random seed.

We don't have anywhere for “state” in Haskell.

We can fake it: introduce a “random generator state” object and pass it around as an argument to everything that generates random numbers.

Haskell does this with `StdGen` instances defined in the `System.Random` module.

```hs
import System.Random
```

But if we have a “random number state”, we need to update it. We can only pass the state as an argument and receive the new state in the return value.

```hs
threeRand :: [Int]
threeRand =
    let gen0 = mkStdGen 1234 -- gen0 :: StdGen
        (rand0, gen1) = randomR (1, 100) gen0
        (rand1, gen2) = randomR (1, 100) gen1
        (rand2, _)    = randomR (1, 100) gen2
    in [rand0, rand1, rand2]
```

Bad things: always uses random seed 1234; lots of managing the `StdGen` instances.

If we want to have an unpredictable seed for the random number generator, we need to look for one: the current time, or network traffic, or processor's true-random value generator. All of those are external state that we have to read.

Read from the outside world: an IO operation. There is a `newStdGen` function that returns a well-seeded random number generator (`StdGen`).

```hs
newStdGen :: IO StdGen
```

We need to work with the `IO` monad to get the `IO StdGen` instance, but can do everything else with the unwrapped `StdGen`.

```hs
threeRand' :: IO [Int]
threeRand' = do
    gen0 <- newStdGen
    let
        (rand0, gen1) = randomR (1, 100) gen0
        (rand1, gen2) = randomR (1, 100) gen1
        (rand2, _)    = randomR (1, 100) gen2
    return [rand0, rand1, rand2]
```

There is a convenience function `randomRs` that generates a infinite list of random values in a range, and we can have much nicer code:

```hs
threeRand'' :: IO [Int]
threeRand'' = do
    gen0 <- newStdGen
    return $ take 3 $ randomRs (1, 100) gen0
```

Another example: generate a bunch of random numbers, and print an ASCII-art histogram to see the distribution. We can generate the random number as before:

```hs
import System.Random

-- generate n random integers
randInts :: Int -> Int -> Int -> IO [Int]
randInts n minval maxval = do
    gen <- newStdGen
    return $ take n $ randomRs (minval, maxval) gen
```

We can generate the histogram in a purely-functional way. It's easier if we do, so we will.

```hs
-- convert a list of values into a histogram
histogram :: (Enum a, Eq a, Ord a) => [a] -> [String]
histogram vals = bars
    where
    counts = [length $ filter (==i) vals
              | i <- [(minimum vals)..(maximum vals)]]
    bars = [take n $ repeat 'X' | n <- counts]
```

Then we can combine them, doing as little monad work as possible.

```hs
-- print histogram of randomly-generated values
printHisto :: IO ()
printHisto = do
    vals <- randInts 1000 1 20
    let bars = histogram vals
    mapM_ putStrLn bars
```

Aside: `mapM_` applies the monad action to each list element.

Full code for this example.

Haskell Tips

A few things for the end of our Haskell exploration...

- Declare types (`::`), especially with recursive functions. Will get you much better error messages meaning “type isn't what you declared” instead of “types don't match”.
- Don't put function definitions in `let`/`where` right away. You can't test those separately.

- Think small.
  - Don't write long expressions: they are too difficult to understand/debug.
  - Write small functions that can be easily checked and debugged.
  - Then combine them.
- Don't repeat yourself. If you feel the urge to copy-and-paste some expression, put it in a `let`/`where`. Better code, and should only get calculated once.

Functional Programming Context

We have looked at one functional language. Haskell is a good example: purely functional, well-designed.

There are others. Generally common to them: pure functions, recursion, first-class functions. Not universal: lazy evaluation, complete lack of state.

Many languages that aren't truly “functional” have borrowed some ideas: use of pure functions, list comprehensions, more use of recursion, pattern matching, partial function application, type inference, lazy evaluation.

The line between functional and non-functional languages can be blurry.

Functional + Imperative

Much of what we have learned about functional programming can be applied to non-functional languages.

Many of the idioms from Haskell can be translated to other languages. Many constructs in modern programming languages (and libraries) may make more sense with some Haskell experience.

[i.e. This was all worth it, even if you never again write Haskell.]

A filter and map in Ruby:

```rb
[1,2,3].select{|x| x!=2}.collect{|x| x*10} == [10, 30]
```

A list comprehension in Python:

```py
[x*10 for x in [1,2,3] if x!=2] == [10, 30]
```

Also, Python 3's functools module and itertools module.

LINQ in C# blends these ideas with SQL ideas:

```cs
int[] numbers = { 1, 2, 3 };
var results = from n in numbers
              where n != 2
              select n * 10;
```

These more functional coding styles are often easier to read/write (and sometimes faster) than the equivalent `for` loop.

Maybe more important: you have gotten used to writing code with **pure functions**. In imperative lanagues, you can create non-pure functions.

- Can have side-effects: `print` or do other I/O, modify argument objects, etc.
- Can depend on more than arguments: global variables, file contents, etc.

But non-pure functions are often harder to test and debug.

You have to control/analyze not only function arguments, but broader state. State is much harder to control: you have to examine any code that modifies it.

Calling non-pure functions concurrently isn't easy. (e.g. Python's `multiprocessing.Pool.map`)

Solution: write pure functions as much as possible.

Writing pure functions in an imperative language:

- Pass in all info you need in arguments.
- Don't modify external state (e.g. don't change arguments or external variables).
- Return whatever results you have.
- Write small, pure, easy-to-test functions.

... as much as possible.

There are going to be cases there that's not realistic advice. If you're writing non-pure functions:

- Be clear about what parts of your code are non-pure (in comments or docs); minimize those parts.
- Be as “local” as possible: accessing or modifying class properties is still better than working with global variables.

The overall message: functional code is often better code, no matter what language you're using.

There are places in imperative languages where you must write (almost?) pure functions.

- task queues (Celery, Resque) where you ask that some code be evaluated later in a job queue: code has to be essentially pure since it is run in a separate process from where it's called.
- distributed computation (e.g. MapReduce, Spark, Flink) where your computation can be spread across many servers with no/little shared state.

Imperative languages are generally strictly evaluated (not lazy), but there are places where lazy evaluation has been introduced because it made sense. e.g.

- the Spark big data framework (for Python, Scala, Java, R) does lazy evaluation on all of its data structures;
- the .NET `System.Lazy<T>` class;
- generators in Python.