Haskell Workshop 2014

Code and walkthrough for the Fall 2014 Hackers @ Berkeley Haskell workshop

Prerequisites

This workshop assumes understanding of functional programming at the level of CS 61A. If you have not taken or are currently taking 61A, it may be hard to follow along.

You should have already installed the Haskell Platform. If you have not, please do so by following the instructions at http://www.haskell.org/platform/

What You Will Learn

I'm not going to make you a Rockstar Haskell Ninja Jedi in two hours. This workshop is, as I said, geared specifically toward those who have taken 61A but have never seen Haskell before. The implications of this are twofold:

1. Much of the focus of most introductory Haskell material is on things that you are already well familiar with, in particular higher order functions and recursion. We don't have to bother learning the ideas governing these concepts from the ground up, so we will be able to get into more unique aspects of the language and solving problems in a purely functional paradigm fairly quickly.
2. As a consequence of that point, you're not going to have a lot of practice with the Haskell syntax by the time we do get into more interesting things. For that reason, the code we go through may be somewhat hard to follow the first time. If there's an overwhelming response of dissatisfaction with this, I'll slow down at the cost of not getting to everything planned.

My goal then is not that you can sit down after the workshop and come up with all of this code again without any reference, but that, should you be intrigued enough to try to continue learning Haskell, you know where to look for further materials and you have a couple starting points for things to hack on that are both familiar and sufficiently interesting. You can look up the syntax as you need to (and if you're anything like me, you'll be doing that for at least a couple months).

The Interpreter

GHCi is the Haskell interpreter that comes packaged with the Platform. Open it up in your terminal by typing `ghci`. You should see something like this:

Figure 1: GHCI

By default, the prompt depends on whatever packages you have loaded. `Prelude` is the package of everything that is imported automatically in GHC. If you're working with a bunch of stuff, the prompt can get kind of unwieldy, so you can change it for the duration of your interactive session using `:set prompt " > "` (you can replace that string with whatever you want, but I think the lambda's pretty cute so that's what I'm gonna use). If you want to make that change permanent, add the same line to your `ghci.conf` file as outlined here: https://www.haskell.org/ghc/docs/7.2.2/html/users_guide/ghci-dot-files.html.

The interpreter works pretty much as you'd expect with arithmetic and whatnot. The most immediate thing you'll need to know to play around is the syntax for calling functions. In Haskell, a function call looks like an operator followed by its operands delimited by spaces. Some examples:

```
> max 1 2
2
> exp 1
2.718281828459045
```

Function application takes the highest precedence.

```
> max 1 2 + min 1 2
3
```

In order to have expressions within operands to a function, you must use parentheses.

```
> sin pi/2
6.123233995736766e-17
> sin (pi/2)
1.0
```

Now let's take a look at lists. Much like lists in Scheme, Haskell lists are linked lists that are constructed using the Cons operator, `:`, terminated by the empty list, `[]`. Luckily, they come with some syntactic sugar that makes them look like Python lists. Also, strings (denoted by double quotes) are actually lists of characters (denoted by single quotes) with some additional syntactic sugar. Behold:

```
> 1:2:3:4:5:[]
[1,2,3,4,5]
> 5:[4,3,2,1]
[5,4,3,2,1]
> ['h','e','l','l','o']
"hello"
> 'c':"ello"
"cello"
```

One important note about Haskell lists that make them very different from Python or Scheme is that they are **homogeneously typed**. That is, you can have a list of `Int`s or of `Double`s or of `Char`s or even of lists of `Int`s, but you cannot have a list with elements of different types. If you try this, GHCi will blow up in your face.

```
> ["this", "statement", "is", False]

<interactive>:2:29:
    Couldn't match expected type `[Char]' with actual type `Bool'
    In the expression: False
    In the expression: ["this", "statement", "is", False]
    In an equation for `it': it = ["this", "statement", "is", ....]
```

This is a type error. You're going to see **a lot** of these. They're the reason it will take ten tries to get your code to compile, and they're the reason that once it does compile, it will work the first time. More on that later.

Now that we've got our basic syntax out of the way, let's talk about functions.

User-Defined Functions

Open up your favorite text editor and create a file called `sandbox.hs` (the name's not important, call it `sandwich.hs` if you'd rather. As long as it starts with

`sand`). In Haskell, like most languages, `=` is used to assign values to variables. Unlike in most languages, though, `=` is not so much of an assignment as it is a definition. The distinction is that all values in Haskell are **completely immutable**. If you say that `x` is five then `x` is five, dammit, no matter what else happens. If you try to change it or provide another definition of `x`, everything will blow up again.

To define a function, we list the name of the function followed by the names of its arguments, space delimited, on the left of the `=`, and we write an expression on the right. Whatever that expression evaluates to is the return value of the function. Let's go through some contrived examples. Type these definitions into `sandbox.hs`:

```
square x = x * x
sumSquares x y = square x + square y
squareSum x y = square (x + y)
```

To play with them, switch back over to your GHCi session and type in this command:

```
> :l sandbox
[1 of 1] Compiling Main             ( sandbox.hs, interpreted )
Ok, modules loaded: Main.
>
```

Let's try them out!

```
> square 4
16
> sumSquares 4 5
41
> squareSum 4 5
81
```

Well. Okay. Let's try to make that a little more exciting. Go back to `sandbox.hs` and define the function `bigness`.

```
bigness x = if x > 50 then "big" else "small"
```

This is Haskell's equivalent of an `if` statement. If you're familiar with ternary expressions in any language, this is the same thing. If not, all that's different is that the entire thing is an expression, which means it must evaluate to something, so the `else` always has to be there. It also means we can use it within other expressions like this:

```
myMood sky = (if sky == "blue" then 'r' else 's') : "ad"
```

So let's play around with these a bit. Remember to `:l sandbox`.

```
 > myMood "blue"
"rad"
 > myMood "black"
"sad"
 > bigness 123
"big"
 > bigness (sumSquares 4 5) + bigness (squareSum 4 5)
<interactive>:5:26:
    No instance for (Num [Char]) arising from a use of `+'
    Possible fix: add an instance declaration for (Num [Char])
    In the expression:
        bigness (sumSquares 4 5) + bigness (squareSum 4 5)
    In an equation for `it':
        it = bigness (sumSquares 4 5) + bigness (squareSum 4 5)
```

Uh oh. Instead of `"smallbig"`, we got a type error. What's up with that?! Doesn't `+` concatenate strings? **NO!** This is right around the point where we need to start talking more seriously about types (if you really want to know, the operator you're looking for is `++`).

The Type System

Haskell is **strongly typed**. This means, unlike Python, a value's type is rigid and can never change. Unlike Javascript, values of one type are not coerced into other types when a function is called on them. This may feel at first like a restriction, but it's actually extremely powerful. The type system is expressive enough that simply writing a type declaration for a function often serves as sufficient documentation. And like I said before, in almost all cases, if you make any mistake, the compiler will catch it and it will be a type error. That means if your code compiles, you probably didn't make any mistakes.

To check the type of something in GHCi, you can do one of two things. Either `set +t`, which will make GHCi print out the type of every value you compute, or do `:t value`, which will just print the type of `value`. I'm gonna stick with the latter.

```
 > :t False
False :: Bool
 > :t 'z'
'z' :: Char
```

```
> :t "derp"
"derp" :: [Char]
```

In the last line we see the syntax for a list of something, in this case `Char`s.

Let's see what types of functions look like.

```
> :t myMood
myMood :: [Char] -> [Char]
```

What GHCi is telling us is that `myMood` is a function that takes a list of characters and returns a list of characters (i.e. a `String`). Pretty straightforward. To add a type declaration to your function, you can type the line that GHCi just gave you into `sandbox.hs` right above the function definition. Note that types are automatically inferred, so of course this is not necessary. Like I said though, type declarations serve as documentation, so it's a good habit to include them as much as possible. `String` and `[Char]` are synonyms, so you can instead type this:

```
myMood :: String -> String
myMood sky = (if sky == "blue" then 'r' else 's') : "ad"
```

and GHC will know what you mean. Type synonyms serve as very terse but clear documentation, and we'll talk more about them later.

Can you give a type signature for `bigness`? One important note: the type `Int` is for 32- or 64-bit signed integers (depending on your architecture). `Integer` represents unbounded integers. Either works here. As an exercise, try adding a type declaration to `bigness` and checking if it compiles by loading `sandbox` in your GHCi session.

There's something totally new that we have to talk about before we can look at functions of more than one argument. Functions in Haskell are curried by default. That means that a function of two arguments is actually a function that takes one argument and returns another function. Hence the type declarations for `sumSquares` should look like this (we'll assume we're working with small integers):

```
sumSquares :: Int -> Int -> Int
```

Type that into `sandbox.hs` in the right place and add type declarations for the other functions in there. We'll see some examples pretty soon of how this can be really useful.

We can treat operators as normal prefix functions by putting them in parentheses. This can be useful both to check the type of an operator in GHCi and to pass things like `*` and `+` into higher order functions.

```
> :t (&&)
(&) :: Bool -> Bool -> Bool
> :t (:)
(:) :: a -> [a] -> [a]
```

What's going on here? The `:` operator is a function that takes in something of some type and a list of that same type and outputs a new list of the same type. `a`, then, is a **type variable**. When you actually use `:` in an expression, Haskell will infer from the context what the type `a` should be. Then `[a]` will be a list of that type. This means that `:` is a **polymorphic function**: it works on more than one type. We can actually add constraints to type variables that allow us to make assumptions about what they can do, which allows us to do some really cool stuff and build really useful domain-specific abstractions. Unfortunately, we don't really have time to get too far into that today. As an example to make what I'm saying more concrete before I abandon the subject, consider the type of `+`:

```
> :t (+)
(+) :: Num a => a -> a -> a
```

Like `:`, `+` works on a type variable `a`. Unlike `:`, there is the constraint that `a` must be a numeric type. To learn more about this, read about typeclasses in one of the linked resources at the end.

The question at hand, now, is how Haskell works with user defined types.

User-Defined Types

Let's consider a complex number example. In rectangular coordinates, a complex number is represented by a pair of the real part and the imaginary part. We haven't talked about tuples yet, but all that really needs to be said about them is that are very strictly typed. What I mean is that a tuple's type is uniquely defined by how many elements it has and what the types of those elements are. `(Int, Int)`, `(Int, Int, Int)`, and `(Int, Double)` are all distinct types.

So our complex numbers can be represented as an ordered pair of type `(Double, Double)`! Brilliant. Now what would we ever want to do with a complex number? Maybe we want to find its magnitude. Okay, so let's define a function in `sandbox.hs`. Note that since we restricted the `square` function before to type `Int -> Int` before, it won't work here.

```
magnitude :: (Double, Double) -> Double
magnitude (re, im) = (re ** 2 + im ** 2) ** (0.5)
```

By the way, the syntax we're using here is a brilliant feature of Haskell, **pattern matching**. In the arguments to the function, we extract the real and imaginary parts by matching the input against the pattern `(re, im)` and making the corresponding bindings within the function.

Okay, this works. But that type declaration for the function is not very descriptive, and we still haven't defined a new type. However, we can define a type synonym for `(Double, Double)` like this:

```
type Complex = (Double, Double)
```

and then we can rewrite the type declaration for `magnitude` like this:

```
magnitude :: Complex -> Double
```

Much better! Although, again, we still haven't defined a new type, only a type *synonym*. It's clear from the type declaration that the function is intended to be used on complex numbers, but since a complex number is represented only by an ordered pair of `Double`s, it will still work on anything else that we choose to represent this way. This could give us problems down the road if we had some unrelated piece of data that we represent as a pair of `Double`s, because then we'd have a set of functions that work on a type they're not intended to be used with. So let's actually define a new type. Delete the type synonym declaration above and add this instead:

```
data Complex = Complex Double Double deriving (Show, Eq)
```

What we've done is define a type `Complex` as well as a data constructor `Complex`. These can be different, as we'll see momentarily. A data constructor is simply a function that takes in some arguments–here they are two `Double`s–and returns a value of the type on the left of the `=` that contains those values. Don't worry about the `deriving` part; here we're just telling Haskell that our `Complex` type can be both printed and equated naively.

Now that we're no longer using a tuple, the pattern matching looks a little different. Change `magnitude` to look like this:

```
magnitude (Complex re im) = (re ** 2 + im ** 2) ** (0.5)
```

A fairly straightforward change. Playing around with our new type may lend some intuition about how that's working. Load `sandbox.hs` into GHCi and try it out. By the way, you can define variables and functions in GHCi using the `let` keyword.

```
> let x = Complex 2 3
> x
Complex 2.0 3.0
> magnitude x
3.605551275463989
```

As an exercise, try writing functions (along with their type declarations) that use pattern matching to extract the real and imaginary components of a complex number. After that, try writing `addComplex` and `multComplex`.

As you can imagine, we'll often want to create data types that are much more complicated than this. We're going to look at a binary tree example soon, but first I want to digress to talk a bit more about pattern matching.

Pattern Matching

You saw pattern matching against tuples and a user-defined type. Another pattern matching technique that can be really nifty is writing multiple definitions for a function. Consider the function `element`, which checks if something is an element of a list. Add this to your sandbox and don't worry about the type declaration for now.

```
element _ []    = False
element x list = if x == head list then True
                 else element x (tail list)
```

We've written a definition for `element` for the base case and a separate one for the recursive case. Pretty neat! It makes the recursion much more visual. That `_` means that we don't care what's there, because nothing could be an element of the empty list.

Since a list with at least one element takes on the form `y : ys` (remember `:` is Cons) where `y` is the first element of the list and `ys` is the rest, we can take this one step further:

```
element _ []     = False
element x (y:ys) = if x == y then True
                   else element x ys
```

Good stuff. Try implementing `listLength` and `butLast`, who both do exactly what they sound like, recursively using pattern matching.

Binary Tree Example

Open `BinTree.hs` from the skeleton folder.

```
data BinTree a = Empty
               | Node a (BinTree a) (BinTree a) deriving (Show)
```

There are several things going on here. The first thing we see is a type variable `a`. That means BinTree is actually a **type constructor**. The type of a binary tree is uniquely determined by the type of its contents, which we've restricted to being homogeneously typed. Examples of concrete types we could make from this are `BinTree Int` and `BinTree Complex` (but make sure you have both files loaded before you try to make one of those).

The next thing we see is that there are two things to the right of the `=`, separated by `|`. `BinTree` has two data constructors. Since `Empty` has no fields, it's actually just a single unique value representing the empty tree. `Node`, on the other hand, is recursively defined as having a left and right subtree. `Node` has three fields. The first, which represents the item stored in that `Node`, has type `a`. The second and third, which are the subtrees, are both binary trees containing items of type `a`, so their type is `BinTree a`.

Let's define a `size` function so we can see that in action. We define "size" to mean the number of elements stored in the tree.

```
size :: BinTree a -> Int
size Empty = 0
size (Node _ left right) = 1 + size left + size right
```

More pattern matching! The algorithm here should be pretty familiar. The size of an empty tree is 0. The size of a tree with a node is 1 greater than the sum of the sizes of the subtrees. Again, that `_` is there because we don't actually care what the element at that node is.

The next example I want to show should also be familiar, though the type may throw you off.

```
treeMap :: (a -> b) -> BinTree a -> BinTree b
```

Yikes! What is that? Let's think about `map` in terms of types. The `map` we're used to from 61A takes a unary function and a list, and returns a list of the results of applying that function to every element of that list. A sensible extension of that into a strongly-typed environment is that the list must be a lists of elements of type `a` and the function must take in a single value of type `a` and return a value of type `b` (could be the same type, could be different). The output, then, would be a list of values of type `b`. And that's what we have here.

Look back at the type declaration and take a moment to digest why this makes sense.

Now let's implement the function.

```
treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMap _ Empty = Empty
treeMap f (Node x left right) = Node (f x) (treeMap f left) (treeMap f right)
```

Mapping any function over the empty tree is the empty tree. Mapping `f` over a `Node` whose element is `x` results in a `Node` whose element is `f x` and whose subtrees are the results of mapping `f` over the subtrees of the original `Node`. Let's play around with our data structure. Two example trees have been provided in the skeleton code. Look for them in `skeleton/BinTree.hs` if you haven't been using the skeleton code. Note the comment at the bottom there about pretty-printed trees and do what it says if you want to. It should still be pretty easy to follow along.

```
 > t1
Node 3 (Node 6 Empty Empty) (Node 2 (Node 4 Empty Empty) Empty)
 > t2
Node 7 (Node 2 (Node 1 Empty Empty) (Node 4 Empty Empty)) (Node 8 Empty Empty)
 > size t1
4
 > let addTwo x = x + 2
 > treeMap addTwo t2
Node 9 (Node 4 (Node 3 Empty Empty) (Node 6 Empty Empty)) (Node 10 Empty Empty)
 > treeMap even t1
Node False (Node True Empty Empty) (Node True (Node True Empty Empty) Empty)
```

Remember when I promised I'd show you why automatic currying is useful? We can actually take any function and underfill its arguments to get another function. Here's why that's cool:

```
 > treeMap (max 5) t2
Node 7 (Node 5 (Node 5 Empty Empty) (Node 5 Empty Empty)) (Node 8 Empty Empty)
 > treeMap (^3) t1
Node 27 (Node 216 Empty Empty) (Node 8 (Node 64 Empty Empty) Empty)
 > treeMap (3^) t1
Node 27 (Node 729 Empty Empty) (Node 9 (Node 81 Empty Empty) Empty)
```

`max 5` evaluates to a function that takes in some number and returns the max of 5 and that number. `^3` and `3^` are functions that cube a number and raise 3 to a number, respectively. You can do that with any binary operator. It's *really* useful.

Before we move on I just want to point out that, knowing this, you can rewrite `square` from back in the `sandbox` like this:

```
square :: Int -> Int
square = (^2)
```

and that's just awesome. Like really awesome. This notion that you can manipulate and define functions without even referring to their arguments is what makes Haskell code able to express ideas so succinctly. I'm probably going to get stuck preaching here during the live workshop, so if you're reading right now, I'm sorry you're missing that. For better or for worse.

Last thing: try to write functions `height` and `flatten` in `BinTree.hs`. `height` finds the length of the longest path from the root to a leaf, and `flatten` returns a list of the elements of a tree in the order they would be visited by a preorder traversal. If you don't know what that means, the output from flattening `t2` should be sorted.

Laziness

One thing I didn't tell you about lists is that there's special syntax to construct ranges out of enumerable types. `[1..10]` and `['a'..'z']` work as you'd expect. `[2,4..20]` makes an arithmetic sequence.

That's all fine and good. What's really neat is that Haskell has no problem with you defining an an infinite list like `[1..]`. How does this work? Haskell is **lazily evaluated**. As it relates to the question at hand, that means **all lists are streams**. You want to find the first integer that's bigger than 1,000,000 when raised to itself? No problem:

```
> let pred n = n^n > 1000000
> head (filter pred [1..])
8
```

Wham! Pretty good. But it doesn't just stop with lists. **Everything** is lazily evaluated. That means **all functions are short-circuiting**. You know how `and` and `or` are special forms in most languages? Not Haskell. If a function knows what it evaluates to by only looking at its first argument, it will look no further. This applies to data structures as well. We could easily create a type to refer to power series, do all sorts of arithmetic with them, and at any time we could request an approximation of the result to arbitrary precision.

That would take too long to try to do in this workshop though. Instead, for a cool example, I'm gonna show you something you've already seen: a recursive fibonacci number stream.

```
> let fibs = 0:1:zipWith (+) fibs (tail fibs)
```

`zipWith (+)` is essentially taking two lists, lining them up, and summing them element-wise. Working in terms of infinite lists is very natural in Haskell. Once we have this, we can do whatever we want with it.

```
> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
> take 10 (filter odd fibs)
[1,1,3,5,13,21,55,89,233,377]
> zip ['a'..'g'] fibs
[('a',0),('b',1),('c',1),('d',2),('e',3),('f',5),('g',8)]
```

It's also really fast. Try typing in `fibs !! 100000` (that's asking for the hundred thousandth element of `fibs`). It's pretty cool.

Doing Stuff (I/O)

Unlike most of haskell, I/O works as a series of commands. If you take a look at io.hs, you see the structure of an I/O block is as follows:

```
f :: IO a
f = do
    x <- action1
    action2 x
    y <- action3
    action4 x y
```

By the type signature, we see that `f` is of type `IO`. However, where as most functions in Haskell are purely functional, an `IO` block uses a `do` statement.

If we look at the example in io.hs:

```
toList :: String -> [Integer]
toList input = read ("[" ++ input ++ "]")

main = do
    putStrLn "Enter a list of numbers (separated by comma):"
    input <- getLine
    print $ sum (toList input)
```

The body of main is a `do` statement, which issues the following commands.

1. First, print to screen "Enter a list of numbers (separated by comma):"
2. Then, use the IO built-in function `getLine` to bind the result of getLine to a variable `input`.
3. Finally, use the user defined function `toList` to create a list from the variable `input`, and print the sum of the list to screen.

Most I/O blocks in Haskell are a lot more complicated than the above example. For more information, and a more comprehensive look at the IO Monad, take a look at Haskell Fast and Hard, or Learn You a Haskell for the Greater Good.