I/O

CIS 194 Week 5
18 February 2015

Further reading

Real World Haskell, Chapter 7 Learn You a Haskell, Chapter 9

The problem with purity

Remember that Haskell is *pure*. This means two primary things:

1. Functions may not have any external effects. For example, a function may not print anything on the screen. Functions may only compute their outputs.

2. Functions may not depend on external stuff. For example, they may not read from the keyboard, or filesystem, or network. Functions may depend only on their inputs—put another way, functions should give the same output for the same input every time.

But—sometimes we *do* want to be able to do stuff like this! If the only thing we could do with Haskell is write functions which we can then evaluate at the ghci prompt, it would be theoretically interesting but practically useless.

In fact, it *is* possible to do these sorts of things with Haskell, but it looks very different than in most other languages.

The `IO` type

The solution to the conundrum is a special type called `IO`. Values of type `IO a` are *descriptions of* effectful computations, which, if executed would (possibly) perform some effectful I/O operations and (eventually) produce a value of type `a`. There is a level of indirection here that's crucial to understand. A value of type `IO a`, *in and of itself*, is just an inert, perfectly safe thing with no effects. It is just a *description* of an effectful computation. One way to think of it is as a *first-class imperative program*.

As an illustration, suppose you have

```
c :: Cake
```

What do you have? Why, a delicious cake, of course. Plain and simple.

By contrast, suppose you have

```
r :: Recipe Cake
```

What do you have? A cake? No, you have some *instructions* for how to make a cake, just a sheet of paper with some writing on it.

Not only do you not actually have a cake, merely being in possession of the recipe has no effect on anything else whatsoever. Simply holding the recipe in your hand does not cause your oven to get hot or flour to be spilled all over your floor or anything of that sort. To actually produce a cake, the recipe must be *followed* (causing flour to be spilled, ingredients mixed, the oven to get hot, *etc.*).

In the same way, a value of type `IO a` is just a "recipe" for producing a value of type `a` (and possibly having some effects along the way). Like any other value, it can be passed as an argument, returned as the output of a

function, stored in a data structure, or (as we will see shortly) combined with other `IO` values into more complex recipes.

So, how do values of type `IO a` actually ever get executed? There is only one way: the Haskell compiler looks for a special value

```
main :: IO ()
```

which will actually get handed to the runtime system and executed. That's it! Think of the Haskell runtime system as a master chef who is the only one allowed to do any cooking.

If you want your recipe to be followed then you had better make it part of the big recipe (`main`) that gets handed to the master chef. Of course, `main` can be arbitrarily complicated, and will usually be composed of many smaller `IO` computations.

So let's write our first actual, executable Haskell program! We can use the function

```
putStrLn :: String -> IO ()
```

which, given a `String`, returns an `IO` computation that will (when executed) print out that `String` on the screen. So we simply put this in a file called `Hello.hs`:

```
main = putStrLn "Hello, Haskell!"
```

Then typing `runhaskell Hello.hs` at a command-line prompt results in our message getting printed to the screen! We can also use `ghc --make Hello.hs` to produce an executable version called `Hello` (or `Hello.exe` on Windows).

GHC looks for a module named `Main` to find the `main` action. If you omit a module header on a Haskell file, the module name defaults to `Main`, so this often works out, even if the filename is not `Main.hs`. If you wish to use a module name other than `Main`, you have to use a command-line option when calling `ghc` or `runhaskell`. Say you have a file `Something.hs` that looks like

```
module Something where
main :: IO ()
main = putStrLn "Hi out there!"
```

You can compile that with `ghc --make -main-is Something Something.hs`. Note the double dashes with `--make` but only a single dash with `-main-is`.

There is no `String` "inside" an `IO String`

Many new Haskell users end up at some point asking a question like "I have an `IO String`, how do I turn it into a `String`?", or, "How do I get the `String` out of an `IO String`"? Given the above intuition, it should be clear that these are nonsensical questions: a value of type `IO String` is a description of some computation, a *recipe*, for generating a `String`. There is no `String` "inside" an `IO String`, any more than there is a cake "inside" a cake recipe. To produce a `String` (or a delicious cake) requires actually *executing* the computation (or recipe). And the only way to do that is to give it (perhaps as part of some larger `IO` value) to the Haskell runtime system, via `main`.

Sequencing `IO` actions

It would all be a little silly if a Haskell program could do only one thing – the thing in the `main` action. We need a way of doing one thing and then the next. Haskell provides a special notation for sequencing actions, called `do` notation. `do` notation is actually very powerful and can be used for wondrous things beyond sequencing I/O actions, but its full power is a story for another day (perhaps several other days).

Here is an action that uses `do` notation to accomplish very little. I'm not naming it `main`, so it can only be accessed from within GHCi, but that's OK for our purposes.

```
sillyExchange :: IO ()
sillyExchange = do
  putStrLn "Hello, user!"
  putStrLn "What is your name?"
```

```
  name <- getLine
  putStrLn $ "Pleased to meet you, " ++ name ++ "!"
```

IO types

Before unpacking that example, it's helpful to look at some types. (Gee, in Haskell, it's *always* helpful to look at some types.)

First, let's start with `()`. The `()` type is pronounced "unit" and has one value, `()`. It's as if it was declared with

```
data () = ()
```

though that's not valid Haskell syntax. `()` is a pretty silly type at first: it conveys absolutely no information, because it has only one constructor that takes no arguments. But, that's exactly what we need in certain I/O actions: `sillyExchange` is an I/O action that produces no (interesting) value at the end. Haskell insists that it has to produce *something*, so we say it produces `()`. (If you squint at `()`, it looks a little like `void` from C/C++ or Java.)

Here are some types:

```
putStrLn :: String -> IO ()
getLine  :: IO String
```

We've seen uses of `putStrLn` before. When sequencing actions with `do` notation, each "bare" line (lines that don't have a `<-` in them) must have type `IO ()`. Happily, `putStrLn "foo"` indeed has type `IO ()`. These actions get performed in order when processing a `do` block.

`getLine`, on the other hand, has type `IO String`. That means that `getLine` is an action that produces a `String`. To get the `String` out of `getLine`, we use `<-` to bind a new variable `name` to that `String`. Here's the catch: you can do this *only* in a `do` block defining an `IO` action. There's no useful way to run `getLine` in code that's not part of an `IO` action. Trying to do this is like getting the cake out of the cake recipe – it's very silly indeed.

It's important to note that `name <- getLine` does not have a type; that is not a Haskell expression. It's just part of the syntax of `do` notation. You can't include `name <- getLine` as part of some larger expression, only as a line in a `do` block.

A slightly larger example

```
jabber :: IO ()
jabber = do
  wocky <- readFile "jabberwocky.txt"
  let wockylines = drop 2 (lines wocky)  -- discard title
  count <- printFirstLines wockylines
  putStrLn $ "There are " ++ show count ++ " stanzas in Jabberwocky."

printFirstLines :: [String] -> IO Int
printFirstLines ls = do
  let first_lines = extractFirstLines ls
  putStr (unlines first_lines)
  return $ length first_lines

extractFirstLines :: [String] -> [String]
extractFirstLines []         = []
extractFirstLines [_]        = []
extractFirstLines ("" : first : rest)
  = first : extractFirstLines rest
extractFirstLines (_ : rest) = extractFirstLines rest
```

There's a bunch of interesting things in there:

1. `readFile :: FilePath -> IO String`, where `type FilePath = String`. This function reads in the entire contents of a file into a `String`.

2. `let` statements within `do` blocks. It would be awfully silly if all of the pure programming we have covered were unusable from within `do` blocks. The `let` statement in a `do` block allows you to create a new variable bound to a *pure* value. Note the lack of `in`. Remember that when you say `let x = y`, `x` and `y` have the same types. When you say `x <- y`, `y` has to have a type like `IO a`, and then `x` has type `a`.

3. `return :: a -> IO a`. If you need to turn a pure value into an I/O action, use `return`. `return` is a regular old function in Haskell. It is *not* the same as `return` in C/C++ or Java! Within an I/O action, `let x = y` is the same as `x <- return y`, but the former is vastly preferred: it makes the purity of `y` more obvious.

There are many functions that you can use to do I/O. See the family of modules starting with `System.`, and in particular, `System.IO`.

More Types!

Record Syntax

Suppose we have a data type such as

```
data D = C T1 T2 T3
```

We could also declare this data type with *record syntax* as follows:

```
data D = C { field1 :: T1, field2 :: T2, field3 :: T3 }
```

where we specify not just a type but also a *name* for each field stored inside the `C` constructor. This new version of `D` can be used in all the same ways as the old version (in particular we can still construct and pattern-match on values of type `D` as `C v1 v2 v3`). However, we get some additional benefits.

1. Each field name is automatically a *projection function* which gets the value of that field out of a value of type `D`. For example, `field2` is a function of type

   ```
   field2 :: D -> T2
   ```

   Before, we would have had to implement `field2` ourselves by writing

   ```
   field2 (C _ f _) = f
   ```

   This gets rid of a lot of boilerplate if we have a data type with many fields!

2. There is special syntax for *constructing*, *modifying*, and *pattern-matching* on values of type `D` (in addition to the usual syntax for such things).

   We can *construct* a value of type `D` using syntax like

   ```
   C { field3 = ..., field1 = ..., field2 = ... }
   ```

   with the `...` filled in by expressions of the right type. Note that we can specify the fields in any order.

   Suppose we have a value `d :: D`. We can *modify* `d` using syntax like

   ```
   d { field3 = ... }
   ```

   Of course, by "modify" we don't mean actually mutating `d`, but rather constructing a new value of type `D` which is the same as `d` except with the `field3` field replaced by the given value.

   Finally, we can *pattern-match* on values of type `D` like so:

   ```
   foo (C { field1 = x }) = ... x ...
   ```

   This matches only on the `field1` field from the `D` value, calling it `x` (of course, in place of `x` we could also put an arbitrary pattern), ignoring the other fields.

ByteStrings

Haskell's built-in `String` type is a little silly. Sure, it's programmatically convenient to think of `String`s as lists of characters, but that's a terrible, terrible way to store chunks of text in the memory of a computer. Depending on an application's need, there are several other representations of chunks of text available.

The `ByteString` library helpfully (?) uses many of the same names for functions as the `Prelude` and `Data.List`. If you just `import Data.ByteString`, you'll get a ton of name clashes in your code. Instead, we use

```
import qualified Data.ByteString as BS
```

which means that every use of a `ByteString` function (including operators) must be preceded by `BS`. Thus, to get the length of a `ByteString`, you use `BS.length`. It's pretty annoying to have to qualify the type `ByteString`, so we often also include:

```
import Data.ByteString (ByteString)
```

This allows you to refer to the type as `ByteString` instead of `BS.ByteString`.

When working with non-`String` strings, it is still very handy to use the `"..."` syntax for writing literal values. So, GHC provides the `OverloadedStrings` extension. This works quite similarly to overloaded numbers, in that every use of `"blah"` becomes a call to `fromString "blah"`, where `fromString` is a method in the `IsString` type class. Values of any type that has an instance of `IsString` can then be created with the `"..."` syntax. Of course, `ByteString` is in the `IsString` class, as is `String`.

A consequence of `OverloadedStrings` is that sometimes GHC doesn't know what string-like type you want, so you may need to provide a type signature. You generally won't need to worry about `OverloadedStrings` as you write your code for this assignment, but this explanation is meant to help if you get strange error messages.

Unlike `String`s, which are sequences of unicode characters, `ByteString`s are sequences of (more traditional) 8-bit characters. The type of 8-bit characters in Haskell is `Word8`, which is essentially an 8-bit unsigned integer. This is kind of a double edged sword. On the one hand, you can't use character literals like `'a'` when dealing with `ByteString`s (although you can still use string literals!). On the other hand, `Word8`s are instances of `Num` and `Integral`, so you can use all of your favorite numeric functions on them! This will come in handy in this week's homework.

IO is a Functor!

Lets say that we wanted to read a `ByteString` from `stdin`, but we want the result as a list of words instead of a single `ByteString`. We could define the following function using `do` notation:

```
getWords :: IO [ByteString]
getWords = do
  ln <- BS.getLine
  return $ BS.split 32 ln -- 32 is the ASCII code for ' '
```

Notice that not much is going on here, but we are using 3 lines to define the function. All we are doing is getting an `IO` value, applying a function to it, and then returning it into `IO` type. Using `Functor`s, we can make this function much cleaner!

```
getWords' :: IO [ByteString]
getWords' = BS.split 32 <$> BS.getLine
```

This simply maps the (pure) splitting operation over the result of the IO action `BS.getLine`.

Generated 2015-02-16 11:25:17.16751