IO

CIS 194 Week 8
11 March 2013

Suggested reading:

- LYAH Chapter 9: Input and Output
- RWH Chapter 7: I/O

The problem with purity

Remember that Haskell is *lazy* and therefore *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).

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`.

Combining `IO`

As should be clear by now, we need a way to *combine* `IO` computations into larger ones.

The simplest way to combine two `IO` computations is with the `(>>)` operator (pronounced "and then"), which has the type

```
(>>) :: IO a -> IO b -> IO b
```

This simply creates an `IO` computation which consists of running the two input computations in sequence. Notice that the result of the first computation is discarded; we only care about it for its *effects*. For example:

```
main = putStrLn "Hello" >> putStrLn "world!"
```

This works fine for code of the form "do this; do this; do this" where the results don't really matter. However, in general this is insufficient. What if we don't want to throw away the result from the first computation?

A first attempt at resolving the situation might be to have something of type `IO a -> IO b -> IO (a,b)`. However, this is also insufficient. The reason is that we want the second computation to be able to *depend* on the result of the first. For example, suppose we want to read an integer from the user and then print out one more than the integer they entered. In this case the second computation (printing some number on the screen) will be different depending on the result of the first.

Instead, there is an operator `(>>=)` (pronounced "bind") with the type

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

This can be difficult to wrap one's head around at first! `(>>=)` takes a computation which will produce a value of type `a`, and a *function* which gets to *compute* a second computation based on this intermediate value of type `a`. The result of `(>>=)` is a (description of a) computation which performs the first computation, uses its result to decide what to do next, and then does that.

For example, we can write a program to read a number from the user and print out its successor. Note our use of `readLn :: Read a => IO a` which is a computation that reads input from the user and converts it into any type which is an instance of `Read`.

```
main :: IO ()
main = putStrLn "Please enter a number: " >> (readLn >>= (\n -> putStrLn (show (n+1))))
```

Of course, this looks kind of ugly, but there are better ways to write it, which we'll talk about in the future.

Record syntax

This material was not covered in lecture, but is provided as an extra resource for completing homework 8.

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.