Clearing Up Week 05 of CIS194

Rushi Shah

30 August 2015

So week five of CIS194 was a roller coaster! It really gets down with the nitty-gritty of Haskell's type system, which is great! But it can also be a bit confusing. I spent a really long time on this week because I know how important it is. With that being said, a lot of time is spent in the assignment pdf, but there isn't THAT much material in the lecture. So here is a bit of clarification on everything mostly just for my own edification. Any benefit you derive is just a side-effect (hah, puns!).

0.1 Why I was confused

In my mind, there were four things and four pieces of syntax.

The four things:

- **Types**
- **Type classes**
- Types are **instances** of type classes
- **Type synonyms** are types that are exactly the same thing as other types

The four pieces of syntax (with examples from the lecture/assignment of CIS194 week 05):

- `data Foo = F Int | G Char`
- `instance Eq Foo where`
- `class Eq a where`
- `type MyChar = Char`
  - The last one was not actually really a part of week 5, but it exists and was floating around in my head, so I have included it.

0.2 How I cleared everything up

This is how things match up with their syntax (along with an explanation of the order things generally are used):

0.2.1 Type and Data

- Define a type for the thing (`MyMaybeInteger`)
  - Use the `data` syntax

```
data MyMaybeInteger = J Integer | N
--J for Just
--N for Nothing
```

- Define a type class for what attributes the thing should have (`MyEq`)
  - Use the `class` syntax

```
class MyEq a where
  equal :: a -> a -> Bool
  notEqual :: a -> a -> Bool
```

- After you have both of those, constrain the type to the type class
  - Use the `instance` syntax

```
--Read as ``an instance of MyEq is MyMaybeInteger''
instance MyEq MyMaybeInteger where
  equal N N = True
  equal (J a) (J b) = a == b
  equal _ _ = False
  notEqual a b = not (equal a b)
```

- Side note: if you want a type synonym, use the `type` syntax.

I guess what really got me mixed up was that to define a new type, you use the `data` syntax, and the `type` syntax is just for type synonyms.

```
type MyInteger = Integer
```

0.3 Conclusion

So there you have it, a bit more clarification on Week 05 of CIS194. It will definitely take me a while to wrap my head around Haskell's type system, but I am excited because it seems very mature and I can't wait to discover its intricacies. To keep track of my efforts on CIS194, check out the github repo and week 05 specifically.

0.4 Post Note: `newtype`

It's been a while since I originally wrote this post, but it has come to my attention that there is actually one more thing (not explicitly mentioned in Week 05) that needs to be mentioned. The **newtype** syntax is very similar to the data syntax.

> The syntax and usage of newtypes is virtually identical to that of data declarations - in fact, you can replace the newtype keyword with data and it'll still compile, indeed there's even a good chance your program will still work. The converse is not true, however - data can only be replaced with newtype if the type has exactly one constructor with exactly one field inside it.

Basically "if you want to declare different type class instances for a particular type, or want to make a type abstract, you can wrap it in a newtype". A good example is using `newtype Email = Email String`. The idea behind that is that your code is more expressive and it adds a layer of typesafety (if a function is expecting an Email, you can't pass it any old string).

But according to this eloquent rant on what's wrong with newtypes, that is only a false sense of security. The gist of the article is that doing something like `newtype Email = Email String` only makes you think your code is more safe because anybody (including you) can wrap a string that is clearly not an email into the newtype and assume it is one which brings you back to square one. The moral of the story is that although newtypes might help, and are useful, they aren't foolproof.