Rapid Applications Prototyping in Haskell

Workshop Session 3

2018-08-02

Objectives

- Understand what we mean by a Higher Kinded Type
- Introduction to the stock higher kinded types, [], IO, Maybe, Identity, (,), Either, (->)
- Introduction to higher kinded type classes: Functor and Foldable
- Interact with YAML and error handling.

Higher-Kinded Types

We know how to check the type of a value with `:t`. Just as values have types, types have kinds.

We can check the kind of a type with `:k`.

Types with kind `*` we call simple or 'concrete' types. We've seen many of these `Int`, `String`, `Bool`, `Ordering`, `[Bool]`. These are types that are completely specified.

Higher-Kinded Types

Some type constructors aren't completely specified. They take types to complete them. For example, the list `[]` constructor itself, requires a type to complete it and turn it into a concrete type.

```
:k []
[] :: * -> *
```

`IO` is another example. The type constructor `IO :: * -> *` is higher kinded. The type `IO String` is concrete.

Maybe

Let's take a look at the Maybe type with `:info Maybe`

```
data Maybe a = Nothing | Just a
```

Think of this like a list that has a maximum size of 1. It's either Nothing, or a single value indicated by 'Just a'

Check we know what we mean by the following statements in the interpreter

```
:t Nothing
:t Just "foo"
:t Just
:k Maybe
:k Maybe String
```

Pair

The product pair type is higher-kinded, but it takes two types. Take a look at `:info (,)`

```
data (,) a b = (,) a b
```

What does all of this mean?

```
:k (,)
(,) :: * -> * -> *
:k (Int, String)
(Int, String) :: *
:k (,) Int
(,) Int :: * -> *
```

Identity

Identity is just a box that contains exactly one element.

```
data Identity a = Identity a
:k Identity
Identity :: * -> *
```

Either

The coproduct `Either` is higher-kinded and takes two types.

```
data Either a b = Left a | Right b
```

We can make values of type 'Either a b' by using the Left and Right constructors.

```
let x = Left 5 :: Either Int String
let y = Right "foo" :: Either Int String
```

What does this line mean in `:info Either`?

```
instance (Eq a, Eq b) => Eq (Either a b)
```

The function arrow (->)

The function arrow (->) is acually a higher kinded type. It takes two types `a` and `b` and returns the function type `a -> b`. It's :info definitition is a little wonky for internal reasons but it works the same.

For a fixed r, we say the type `((->) r)` is the type of all the functions emanating from r. Also called a hom-functor. This is useful later.

IO

IO is a higher kinded type specially reserved for IO values.

You can see this in functions like `readFile`

```
:t readFile
readFile :: FilePath -> IO String
```

IO type contains exactly one value.

Maps and HashMaps

Maps and HashMaps are higher kinded types that takes a key type, and a value type.

```
:k HashMap
HashMap :: * -> * -> *
```

Their definition is also wonky so don't worry about it too much.

Functors/Foldables

Higher kinded types can have type classes associated with them which allow us to talk about their capabilities.

The simplest of these are `Functor` and `Foldable`

Functors

The functor class has the following definitions

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

fmap is a generalisation of the more familiar `map :: (a -> b) -> [a] -> [b]` We abstract the list part into a more general higher kinded container. We can use fmap in place of map for lists, and we can use it to run a function on the values of other higher kinded types. We can see the functors available to use with `:info Functor`

```
fmap (+1) [1,2,3]
```

Functor examples

Maybe, Identity and IO are functors.

```
let x = Just 3 :: Maybe Int
fmap (+1) x
```

Functor examples

(,), Either and HashMap are functors in their right argument, leaving the left fixed.

```
fmap (+1) ("foo", 6)
fmap (+1) Right 5
fmap (+1) Left "foo"
```

Try using fmap with a HashMap.

Making our own functors

```haskell
data V3 a = V3 a a a deriving (Eq, Show)

instance Functor V3 where
  fmap f (V3 x y z) = V3 (f x) (f y) (f z)
```

OR

Add `DeriveFunctor` to your language extensions and:

```haskell
data V3 a = V3 a a a deriving (Eq, Show, Functor)
```

Now we can fmap

```
let v = V3 1 2 3

fmap (*2) v
```

YAML

http://hackage.haskell.org/package/yaml-0.9.0/docs/Data-Yaml.html

Look at the decoding functions for yaml

```
decodeEither'     :: FromJSON a => ByteString -> Either Pars

decodeFileEither :: FromJSON a => FilePath -> IO (Either Pa

decodeThrow       :: (MonadThrow m, FromJSON a) => ByteStrin

decodeFileThrow   :: (MonadIO m, FromJSON a) => FilePath ->
```