Rapid Applications Prototyping in Haskell

Workshop Session 5

2018-09-09

Objectives

- Monads
- Monads
- Monads
- Monads

Natural Transformations

A natural transformation is a function between functors that doesn't change the underlying type variable.

i.e any function **`h`** of the form:

```
h :: f a -> g a
```

Sometimes you might see a formal type constructor like this:

```
type (~>) f g = forall x. f x -> g x
```

Natural Transformations

For example `headMaybe` from `RIO.List` is a natural transformation between `[]` and `Maybe`

```haskell
headMaybe :: [a] -> Maybe a
```

We saw last time that `sequenceA` is a generic natural transformation

```haskell
sequenceA :: (Traversable a, Applicative f) => t (f a) -> f
```

that allowed us to get from a Graph [a] to a [Graph a]

Monads

A monad is a functor equipped with two natural transformations.

```
class Applicative m => Monad m where
   return :: a -> m a
   join   :: m (m a) -> m a
```

`return` allows us to put a single value in the structure.

`join` allows us to collapse two layers of structure to a single layer.

Maybe and []

Are the following joins possible? How are they implemented?

```
join :: Maybe (Maybe a) -> Maybe a

join :: [[a]] -> [a]
```

Maybe and []

```
instance Monad Maybe where

  return x = Just x

  join (Just (Just x)) = Just x
  join _ = Nothing
instance Monad [] where

  return x = [x]

  join :: [[a]] -> [a]
  join = concat
```

Hom Join

What about the hom-functor `(r ->)`

```
return :: (a -> r -> a)

join :: (r -> (r -> a)) -> (r -> a)
```

Effects

Monads give us a way of expressing computational ‘effects’.

An ‘effectful’ function is usually thought of as an Arrow `a -> m b` for some Monad m. We call these arrows ‘Kleisli’ arrows.

Effects

Depending on the type of Monad, we get different effects.

- In the case of `Maybe`, we get an ‘failure’ effect.
- In the case of `[]`, we have a ‘non-determinism’ effect.
- In the case of `IO`, we have ‘real world’ or ‘side’ effects.
- In the case of `(r ->)`, we an ‘implicit config’ or ‘reader’ effect.
- In the case of `Either s`, we get an ‘exception’ effect.
- In the case of `(,s)` for some monoid `s`, we get a ‘logging’ effect.
- In the case of `(s -> (,s))` for some `s` we get a ‘state’ effect.

Kleisli Composition

We call arrows of the form `a -> m b` for some Monad m a "Kleisli Arrow in m".

Because we have the join operation, we are able to do this:

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

This is called Kleisli composition, and is defined like so:

```
f >=> g = join . fmap g . f
```

Kleisli Composition

Kleisli composition feels a lot like the dot operator, but unlike the dot operator there is implicit work done due to the join operation.

Sequencing Effects

How does Kleisli composition let us sequence effects?

```
(>=>) :: (a -> Maybe b) -> (b -> Maybe c) -> (a -> Maybe c
```

If we sequence two Maybe arrows, the result will be a Maybe arrow which returns Nothing if either of the two previous arrows returned Nothing, but the first arrow must evaluate.

Notice that the order of execution of these arrows is actually guaranteed. If the first computation returns Nothing, we do not have a `b` with which to call the second function. The second function will only execute if the first function returns a Just value.

Sequencing Effects

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

`readFile` is a Kleisli Arrow.

```
readFile :: FilePath -> IO ByteString
```

We can use regular composition `(.)` and Kleisli composition `(>=>)` to get to where we want.

What is this doing?

```
readFile >=> writeFile "out"
```

mapM

mapM runs a Kleisli Arrow along a Traversable.

```
mapM :: (a -> m b) -> t a -> m (t b)
```

```
mapM readFile ["foo.txt", "bar.txt", "baz.txt"]
```

This is a really common use case.

Binding

Equivalently, if we already have an `m a`, and a Kleisli arrow, we can use the bind operation.

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

This is just `(>=>)` where the first arrow has already been evaluated to a result `m a`.

Do notation

Why do we write some variable assignments as `let y = f x` and some as `y <- f x`?

```haskell
main :: IO ()
main = do
  x <- readFile "foo.txt"
  writeFile "bar.txt" x
```

This implictly desugars as `readFile "foo.txt" >>= writeFile "bar.txt"`

This allows you to write an sequence of Kleisli arrows as if it were an imperative program.

Do notation

Remember, the behaviour of the composition will be very idiomatic to the specific monad you're working in. For example, lists have multiple values in, and so the `<-` notation draws out all of them in parallel. This is basically a list comprehension.

```
deck :: [Card]
deck = do
  s <- [Heart..]
  v <- [Two..]
  return (s, v)
```

Do notation

Here's a function which might throw an exception.

```
divE :: Double -> Double -> Either String Double
divE a b = if b == 0 then Left "DivByZero error" else Right

foo :: Double -> Either String Double
foo a = do
  x <- divE a 3
  y <- divE x 0
  z <- divE y 4
  return z
```

Reader Monad

The Reader monad is just a wrapper around the hom functor.

```
newtype Reader r a = Reader { runReader :: r -> a }
  deriving (Functor, Applicative, Monad)
```

We can use this to implicitly pass around a context to subroutines, similar to how objects with a `self` work. We can acces the context with the `ask :: m r` function.

`Reader` is a `Functor` in `a`, and a `Monad` so we can chain functions with this implicit `self` like so:

```
(>=>) :: (a -> Reader r b) -> (b -> Reader r c) -> (a -> Re
(>>=) :: Reader r a -> (a -> Reader r b) -> Reader r b
```

Reader Monad

```haskell
data Config = Config {
  port :: Int
, hostname :: String
} deriving (Eq, Show)

foo :: Reader Config Bool
foo = do
  x <- ask
  return (port x == 3000)
```

runReader

```
runReader :: Reader r a -> r -> a
```

```
let x = Config 3000 "localhost"
runReader foo x
```

This isn't super useful by itself over just writing a function `Config -> IO Bool`, but we can compose it with other effects.

ReaderT Transformer

A monad transformer is a Monad that can compose with the effects of another monad.

Here is `ReaderT`, it's the same as `Reader` except it wraps a Kleisli Arrow from some fixed `r` for some other Monad `m`. This can be any monad, including other monad transformers.

```
newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }
```

The approach RIO takes is this:

```
type RIO r a = ReaderT r IO a
```

with a corresponding `runRIO :: RIO r a -> r -> a`

So a program is just a composition of the hom monad over some fixed context, and the IO monad.