Rapid Applications Prototyping in Haskell

Workshop Session 4

2018-09-02

Objectives

- Recap basic types and type classes.
- Using RIO and stack to write standalone scripts.
- More higher-kinded typeclasses, Foldable, Traversable, Applicative.
- Quick look at algebraic-graphs.

Quick Recap

- `Values` are things we want to compute.
- `Types` are ways of classifying a range of values.
- `Functions` are ways of turning values of one type into values of another type
- `Type Classes` are ways of classfying types by an interface of functions available for that type.
- `Higher Kinded Types` make new types out of existing types or “functions at the type level”

Simple scripts

Make a new blank file “foo.hs” and put

```
#!/usr/bin/env stack
-- stack runhaskell --resolver lts-12.13 --package rio

{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}

import RIO

main = runSimpleApp $ do
  logInfo $ "Hello World"
```

Now you can `chmod a+x foo.hs` and run it with `./foo.hs`.

Remember you can `export RIO_VERBOSE=true` to add color and timestamps.

Shelling out

You can use the proc functiona to shell out.

```
import RIO
import RIO.Process

main = runSimpleApp $ do
  logInfo $ "Hello World"
  proc "touch" ["tiny kitten"] (runProcess_)
  (c, s, e) <- proc "uname" [] (readProcess)
  logInfo $ displayShow $ s
```

Examples

This works really well with some libraries like diagrams, inline-r, clay and lucid.

Here are some links:

- diagrams
- inline-r

Foldables

Foldables are things that can be iterated over and collapsed to a result.

Type `:info Foldable`

```
class Foldable (t :: * -> *) where
  foldr :: (a -> b -> b) -> b -> t a -> b
  ...
```

The first argument is a binary operation, like (+), (++) or (:)

The second argument is an initial value to start with (in case the Foldable is empty).

The third argument is the foldable, with some `a`s inside.

Foldable

`Maybe` and `[]` are both Foldable, so we can do things like:

```
sumList :: [Int] -> Int
sumList = foldr (+) 0
```

But why do this for just lists? We can sum any foldable with the same definition

```
sumFoldable = foldr (+) 0
```

What's the type of `sumFoldable`?

DeriveFoldable

We can derive Foldable in the same way we derived Functor for well behaved data structures

```
{-# LANGUAGE DeriveFoldable #-}
{-# LANGUAGE DeriveFunctor  #-}

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

Algebraic Graphs

algebraic-graphs is a great library that defines the following data structure.

```
data Graph a =
  Empty
| Vertex a
| Connect (Graph a) (Graph a)
| Overlay (Graph a) (Graph a)
deriving (Functor, Foldable, Show, Traversable)
```

Graph is higher kinded, takes a type parameter ‘a’, which allows us to talk about a “graph of ’a’s”

This probably looks like a perculiar way to describe a graph. What values can a graph like this be?

Graphs are naturally Functors and Foldables.

Algebraic Graphs

```haskell
#!/usr/bin/env stack
{- stack runhaskell --resolver lts-12.13
  --package rio --package algebraic-graphs -}

{-# LANGUAGE NoImplicitPrelude #-}

import RIO
import Algebra.Graph

myGraph :: Graph String
myGraph = overlay
  (vertex "spatula")
  (connects $ [vertex "quux",
               vertices ["foo", "bar", "baz"]])

main = runSimpleApp $ do
  logInfo $ displayShow $ myGraph
  logInfo $ displayShow $ foldr (++) "" myGraph
```

Currying/Uncurrying

We know that haskell has two ways of representing two argument functions.

```
(a, b) -> c
a -> b -> c
```

Haskell provides two functions to switch between these representations:

```
curry :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c
```

This of course has the property that

```
curry . uncurry = id
uncurry . curry = id
```

Applicative Functors

A lax monoidal functor, or "applicative" functor is a functor that can lift functions of multiple arguments.

Remember the signature for fmap `fmap :: (a -> b) -> f a -> f b`

```
class Functor f => Applicative f where
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  (<*>)  :: f (a -> b) -> f a -> f b
```

liftA2 is a two argument analogue of fmap, it lifts a two argument function into the higher kinded type.

The `(<*>)` is pronounced `splat`, sometimes called 'ap', it is defined as `(<*>) = liftA2 id`

`Maybe`, `IO`, `[]`, and the hom functor `(r ->)` are all Applicatives

Applicative Functors

One project euler problem says

"Find the largest palindrome made from the product of two 3-digit numbers."

What are all the products of two digit numbers?

We can use `liftA2` to take normal multiplication of numbers and turn it into a function that multiplies two lists of numbers.

```
liftA2 (*) [100..999] [100..999]
```

Sometimes the "fmap-then-splat" notation is used to the same effect.

```
(*) <$> [100..999] <*> [100..999]
```

Similarly we can lift binary operations to operations which act on two Maybes, or two IO values.

Applicative Functors

Another list example:

```
data Value = Two | Three | Four | Five | Six |
             Seven | Eight | Nine | Ten |
             Jack | Queen | King | Ace
  deriving (Eq, Ord, Enum, Show, Bounded)

data Suit = Hearts | Clubs | Spades | Diamonds
  deriving (Eq, Ord, Enum, Show, Bounded)

type Card = (Value, Suit)

type Desk = [Card]

deck = liftA2 (,) [Two..] [Hearts..]
-- deck = (,) <$> [Two..] <*> [Hearts..]
```

The Hom Functor is Applicative

Let's continue the euler problem.

Recall the hom functor `(r ->)`, or "all of the functions emanating from a type `r`".

Just fill it in algebraically to see what fmap and splat do.

```
fmap :: (a -> b) -> f a -> f b
```

becomes

```
fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

This is just the same as the dot (.) operator, so the `fmap` for the hom functor is just the dot operator (.).

The Hom Functor is Applicative

```
liftA2 :: a -> b -> c -> f a -> f b -> f c
(<*>) :: f (a -> b) -> f a -> f b
```

becomes

```
liftA2 :: a -> b -> c -> (r -> a) -> (r -> b) -> (r -> c)
(<*>) :: (r -> a -> b) -> (r -> a) -> r -> b
```

So what good is this?

IsPalindrome

Let's try to make an `isPalindrome :: String -> Bool` that decides whether or not a string is a palindrome.

```
isPalindrome x = ???
```

IsPalindrome

```
isPalindrome x = x == reverse x
```

Put this into http://pointfree.io/ and see what happens.

IsPalindrome

So equivalently:

```
isPalindrome = (==) <*> reverse
```

Can you say why?

Traversable

A Traversable is just something that is both Functor and Foldable.

```
class (Functor t, Foldable t) => Traversable t
```

This has some interesting effects when you mix it with an applicative.

```
traverse  :: (Traversable t, Applicative f) =>
             (a -> f b) -> t a -> f (t b)

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

Functions of the form `(a -> f b)` for some function f are called "Stars".

traverse takes a `a -> f b` and turns it into a `t a -> f (t b)`

Traversable

sequenceA is just equivalent to traverse id

There's also a function `for` which is just `traverse` with its arguments flipped, and works like a for loop.

```
for :: (Traversable t, Applicative f) => t a -> (a -> f b
```

We'll look more at these when we do monads, for now let's try a simple sequence.

Traversable

Recall our graph earlier `myGraph :: Graph String`

Remember that a `String = [Char]`, so, `myGraph :: Graph [Char]`

Graph is a Traversable, and [] is an Applicative, so this is a t (f a)

So we can sequenceA myGraph

What happens?

Traversable

It turns a `Graph [Char]` into a `[Graph Char]`, or a list of possible Graphs of different characters.

```
sequenceA myGraph
:t sequenceA myGraph
```

How many Graphs has it produced?

```
length $ sequenceA myGraph
```