Rapid Applications Prototyping in Haskell

Workshop Session 2

2018-07-26

Objectives

- Introduction to Constructive Type Theory
- Introduction to Simple Algebraic Data Types: Products and Sum Types
- Introduction to basic Type Classes: Semigroup, Monoid, Eq, Ord and Show
- Run RIO with a specific environment.

Constructive type theory

The Haskell type system has an intuitive shape. We can specify simple types as ranging over a set of values, and we can build more complex types out of existing simple types with algebraic constructions.

We can also talk about classes of types by what we can say it is possible to do with them.

Today we focus on simple types and their type classes in the standard library.

Let's cd into our project from last time and run `stack ghci`, and be sure to `import RIO`

Type Classes

A type class is an interface into a type. We can declare type signatures in the class and algebraic laws that the implementations must satisfy.

Only type signatures are enforced by the compiler. Laws are not enforced by the compiler, they are enforced informally.

When reading or documenting a type class description, the laws should be included as documentation.

A type that is able to implement all signatures of the type class whilst satisfying all of the documented laws is called an `instance` of the class.

Semigroups

`Semigroups` are a class of type with an binary operator `<>`, pronounced ‘smush’.

```
class Semigroup a where
  (<>) :: a -> a -> a
```

that must satisfy the associative law:

```
(x <> y) <> z = x <> (y <> z)
```

Types that implement this interface are those that have a natural way to smush their values together.

Examples of Semigroups

Strings can be smushed together

```
"Hello" <> ", " <> "World!"
```

and in fact, any `[a]`.

```
[1,2,3] <> [4,5,6]
```

And all of the different String-like variants in RIO (ByteString, Text, Utf8Builder)

Examples of Semigroups

RIO has different kind of dictionary types, `Map` and `HashMap`. These are both Semigroups.

Container types like List, Set, Map and HashMap often overload names, so you should import them qualified to avoid name collisions.

```
import qualified RIO.HashMap as HM

let x = HM.fromList [(5, 'a'), (3, 'b')] :: HM.HashMap Int
let y = HM.fromList [(4, 'c'), (2, 'a')] :: HM.HashMap Int


x <> y
```

In the case of two entries colliding, the left hand side takes precedence. (By convention).

```
let z = HM.fromList [(5, 'b')] :: HM.HashMap Int Char
x <> y <> z
```

Monoids

A Monoid is a Semigroup with an special empty element called `mempty`:

```haskell
class Semigroup a => Monoid a where
  mempty :: a
```

that satisfies the identity law:

```haskell
mempty <> x = x <> mempty = x
```

Monoids

Lists and HashMaps are also Monoids. You can see the mempty element for each type by specifying the type of mempty.

```
mempty :: String
mempty :: [a]
mempty :: HashMap Int Char
```

In Monoid language, `(<>)` is also called `mappend`. This is historical.

Diagrams

A good paper on monoids is Brent Yorgey's Monoids : Theme and Variations

https://repository.upenn.edu/cgi/viewcontent.cgi?article=1773&context=cis_papers

Diagrams form a monoid by stacking one on top of the other.

Other type classes

Other common type classes in the standard library are `Eq`, `Ord`, `Show`, `Read`, `Bounded`, `Enum`

You can type for example `:info Eq` to get information on a type class - it's signatures and which types in scope it is defined for.

Eq

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Ord

```
class Ord a where
  (<)      :: a -> a -> Bool
  (<=)     :: a -> a -> Bool
  ...
```

Show

The Show typeclass gives us a way of printing to the screen.

```
class Show a where
  show :: a -> String
```

Algebraic Data Types

Haskell has various ways of constructing new types from old types.

The simplest way is using product and sum (also called coproduct) types.

Products

A product type takes two or more types to form a compound type (much like say a struct in C)

You can do this either anonymously as a pair `(a,b)`, or as a record.

For example

```haskell
data Dragon = Dragon {
  name  :: String
, heads :: Int
}
```

Important note: Note, the `Dragon` on the left is a type constructor, the `Dragon` on the right is a value constructor. This is one of the biggest gotchas for beginners.

Making a Dragon

This actually creates three functions for us.

```
-- Dragon :: String -> Int -> Dragon
-- name   :: Dragon -> String
-- heads  :: Dragon -> Int
```

We can make a dragon with

```
let x = Dragon "Bob" 3
```

Product Lemmas for Eq and Show

If types `a` and `b` can each be compared for equality, then their product (a,b) can also be compared for equality.

```
instance (Eq a, Eq b) => Eq (a, b) where
  (x1, y1) == (x2, y2) = x1 == x2 && y1 == y2
```

If types `a` and `b` can each be printed to String, then their product can be printed to a String.

```
instance (Show a, Show b) => Show (a, b) where
  show (x, y) = "(" <> show x <> "," <> show y <> ")"
```

Try it

```
("foo", 3) == ("foo", 3)
show (3,4)
```

Deriving Eq and Show

Haskell can use this lemma to automatically derive the type class instances for `Eq` and `Show`.

```
data Dragon = Dragon {
  name  :: String
, heads :: Int
} deriving (Eq, Show)
```

This allows us to print our dragons to the screen.

```
let x = Dragon "Bob" 3
let y = Dragon "Rainbow" 4
x
y
x == y
```

Product Lemma for Semigroups

If types `a` and `b` are Semigroups, then their product is a Semigroup

```
instance (Semigroup a, Semigroup b) => Semigroup (a, b) whe
  (x1, y1) <> (x2, y2) = (x1 <> x2, y1 <> y2)
```

Smushing a pair just requires us to smush together the individual components.

Try it

```
("foo", "quux") <> ("bar", "spatula")
```

Product Lemma for Monoids

If types `a` and `b` are Monoids, then their product is a Monoid

```haskell
instance (Monoid a, Monoid b) => Monoid (a, b) where
  mempty = (mempty, mempty)
```

However, Haskell will *not* automatically derive these for our product record type. It's theoretically possible, but you often want something else. In our case, Int isn't a Monoid anyway.

Dragon the Semigroup

Let's define our own Dragon Semigroup instance. Our smushed Dragon will be a Dragon with the names smushed together, and the combined sum of heads of the two dragons.

```
instance Semigroup Dragon where
  Dragon n1 h1 <> Dragon n2 h2 =
    Dragon (n1 <> n2) (h1 + h2)
```

```
let x = Dragon "Bob" 3
let y = Dragon "Rainbow" 4
x <> y
```

The Dragon Monoid

An empty Dragon is a Dragon with no name, and no heads.

```
instance Monoid Dragon where
  mempty = Dragon "" 0
```

There's no point to smush with this dragon.

Void

We have a type with no elements, called Void.

It is declared in Haskell as

```
data Void
```

You can see the data declaration by typing `:info Void`

This is used in one function, the absurd function that you can never call

```
absurd :: Void -> a
```

Unit

The type that contains one element is called unit, written `()`.

Type `:info ()` to see the data declaration.

```
data () = ()
```

In a data declaration, the left hand side indicates the type, where the right hand declares the possible values. Here we see the unit type, `()` has one possible value, also called `()`.

You can also see this by checking the type of `()` to see

```
() :: ()
```

Bool

Look at the data declaration for `Bool` with `:info Bool`, it has two possible values, `False` and `True`

```
data Bool = False | True
```

This is called a sum (or coproduct) type.

Ordering

Look at the data declaration for `Ordering` with `:info Ordering`, it has three possible values, `LT`, `EQ` and `GT`.

```haskell
data Ordering = LT | EQ | GT
```

Logging Dragons

RIO's logInfo takes a Utf8Builder, which is a Monoid. We can get from a `Show a` to a `Utf8Builder` with `displayShow :: Show a => a -> Utf8Builder`.

```haskell
main :: IO ()
main = do
  let x = Dragon "Rainbow" 4
  runSimpleApp $ do
    logInfo $ "Hello, " <> (displayShow . name $ x)
```

runRIO

Instead of `runSimpleApp`, we're going to use runRIO with our own environment that just contains a logFunc.

HasLogFunc

logInfo has the signature

```
logInfo :: (MonadIO m, MonadReader env m, HasLogFunc env, H
```

It's ok if this is jibberish. For now, all we're interested in is making sure The part we're particularly interested in is the fact that the environment is an instance of `HasLogFunc`.

```
class HasLogFunc env where
  logFuncL :: Lens' env LogFunc
```

We won't get into lenses, but think of this as an object getter. We have a way to extract a logFunc from the environment.

LogFunc

A logFunc itself is an instance of `HasLogFunc`. That's helpful. We don't need to do anything.

Making our own log function

```haskell
main :: IO ()
main = do
  let x = Dragon "Rainbow" 4
  logStdout <- logOptionsHandle stdout True
  withLogFunc logStdout $ \lfs ->
    runRIO lfs $ do
      logInfo $ "Hello, " <> (displayShow . name $ x)
```

Tee

```haskell
main :: IO ()
main = do
  let x = Dragon "Rainbow" 4
  withBinaryFile "log.txt" WriteMode $ \logHandle -> do
    logStdout <- logOptionsHandle stdout True
    logFile   <- logOptionsHandle logHandle True
    withLogFunc logStdout $ \lfs ->
     withLogFunc logFile   $ \lfo ->
      runRIO (lfs <> lfo) $ do
       logInfo $ "Hello, " <> (displayShow . name $ x)
```