Если вы видите что-то необычное, просто сообщите мне.

Purescript Part 2: Typeclasses and Monads

In part 1 of this series, we started our exploration of Purescript. Purescript seeks to bring some of the awesomeness of Haskell to the world of web development. Its syntax looks a lot like Haskell's, but it compiles to Javascript. This makes it very easy to use for web applications. And it doesn't just look like Haskell. It uses many of the important features of the language, such as a strong system and functional purity.

If you need to brush up on the basics of Purescript, make sure to check out part 1 again. In this part, we're going to explore a couple other areas where Purescript is a little different. We'll see how Purescript handles typeclasses, and we'll also look at monadic code. We'll also take a quick look at some other small details with operators. In part 3, we'll look at how we can use Purescript to write some front-end code.

For another perspective on functional web development, check out our Haskell Web Series. You can also download our Production Checklist for some more ideas!

TYPE CLASSES

The idea of type classes remains pretty consistent from Haskell to Purescript. But there are still a few gotchas. Let's remember our Triple type from the last part.

```
data Triple = Triple
  { a :: Int
  , b :: Int
  , c :: Int
  }
```

Let's write a simple Eq instance for it. To start with, instances in Purescript must have names. So we'll assign the name tripleEq to our instance:

```
instance tripleEq :: Eq Triple where
  eq (Triple t1) (Triple t2) = t1 == t2
```

Once again, we only unwrap the one field for our type. This corresponds to the record, rather than the individual fields. We can, in fact, compare the records with each other. The name we provide helps Purescript to generate Javascript that is more readable. Take note: naming our instances does NOT allow us to have multiple instances for the same type and class. We'll get a compile error if we try to create another instance like:

```
instance otherTripleEq :: Eq Triple where
  ...
```

There's another small change when using an explicit import for classes. We have to use the class keyword in the import list:

```
import Data.Eq (class Eq)
```

You might hope we could derive the Eq typeclass for our Triple type, and we can. Since our instance needs a name though, the normal Haskell syntax doesn't work. The following will fail:

```
-- DOES NOT WORK
data Triple = Triple
  { a :: Int
  , b :: Int
  , c :: Int
  } deriving (Eq)
```

For simple typeclasses though, we CAN use standalone deriving. This allows us to provide a name to the instance:

```
derive instance eqTriple :: Eq Triple
```

As a last note, Purescript does not allow orphan instances. An orphan instance is where you define a typeclass instance in a different file from both the type definition and the class definition. You can get away with these in Haskell, though GHC will warn you about it. But Purescript is less forgiving.

The way to work around this issue is to define a newtype wrapper around your type. Then you can define the instance on that wrapper.

EFFECTS

In part 1, we looked at a small snippet of monadic code. It looked like:

```
main :: Effect Unit
main = do
  log ("The answer is " <> show answer)
```

If we're trying to draw a comparison to Haskell, it seems as though Effect is a comparable monad to IO. And it sort've is. But it's a little more complicated than that. In Purescript, we can use Effect to represent "native" effects. Before we get into exact what this means and how we do it, let's first consider "non-native" effects.

A non-native effect is one of those monads like Maybe or List that can stand on its own. In fact, we have an example of the List monad in part 1 of this series. Here's what Maybe might look like.

```
maybeFunc :: Int -> Maybe Int


mightFail :: Int -> Maybe Int
mightFail x = do
  y <- maybeFunc x
  z <- maybeFunc y
  maybeFunc z
```

Native effects use the Effect monad. These include a lot of things we'd traditionally associate with IO in Haskell. For instance, random number generation and console output use the Effect monad:

```
randomInt :: Int -> Int -> Effect Int


log :: String -> Effect Unit
```

But there are also other "native effects" related to web development. The most important of these is anything that writes to the DOM in our Javascript application. In the next part, we'll use the

Halogen library to create a basic web page. Most of its main functions are in the Effect monad. Again, we can imagine that this kind of effect would use IO in Haskell. So if you want to think of Purescript's Effect as an analogue for IO, that's a decent starting point.

What's interesting is that Purescript used to be more based on the system of free monads. Each different type of native effect would build on top of previous effects. The cool part about this is the way Purescript uses its own record syntax to track the effects in play. You can read more about how this can work in chapter 8 of the Purescript Book. However, we won't need it for our examples. We can just stick with Effect.

Besides free monads, Purescript also has the purescript-transformers library. If you're more familiar with Haskell, this might be a better starting spot. It allows you to use the MTL style approach that's more common in Haskell than free monads.

SPECIAL OPERATORS

It's worth noting a couple other small differences. Some rules about operators are a little different between Haskell and Purescript. Since Purescript uses the period operator . for record access, it no longer refers to function composition. Instead, we would use the <<< operator:

```
odds :: List Int -> List Int
odds myList = filter (not <<< isEven) myList
  where
    isEven :: Int -> Boolean
    isEven x = mod x 2 == 0
```

Also, we cannot define operators in an infix way. We must first define a normal name for them. The following will NOT work:

```
(=%=) :: Int -> Int -> Int
(=%=) a b = 2 * a - b
```

Instead, we need to define a name like addTwiceAndSubtract. Then we can tell Purescript to apply it as an infix operator:

```
addTwiceAndSubtract :: Int -> Int -> Int
addTwiceAndSubtract a b = 2 * a - b

infixrl 6 addTwiceAndSubtract as =%=
```

Finally, using operators as partial functions looks a little different. This works in Haskell but not Purescript:

```
doubleAll :: List Int -> List Int
doubleAll myList = map (* 2) myList
Instead, we want syntax like this:

doubleAll :: List Int -> List Int
doubleAll myList = map (_ * 2) myList
```

CONCLUSION

This wraps up our look at the key differences between Haskell and Purescript. Now that we understand typeclasses and monads, it's time to dive into what Purescript is best at. In part 3 we'll look at how we can write real frontend code with Purescript!

For some more ideas on using Haskell for some cool functionality, download our Production Checklist! For another look at function frontend development, check out our recent Elm Series!

Revision #1
Created 2022-03-11 16:53:52 UTC by gasick
Updated 2022-03-11 17:11:16 UTC by gasick